NEURAL NETWORKS AS A CONVEX PROBLEM

By

Baha Khalil

A Thesis Presented to the Faculty of the

American University of Sharjah

College of Arts and Science

in Partial Fulfillment

of the Requirements

for the Degree of

Master of Science in

Mathematics

Sharjah, United Arab Emirates

September 2016

Approval Signatures

We, the undersigned, approve the Master's Thesis of Baha Khalil.

Thesis Title: Neural Networks as a Convex Problem

Signature	**Date of Signature** (31/08/2016)
Dr. Dmitry Efimov Assistant Professor Thesis Advisor	
Dr. Suheil Khoury Professor Thesis Committee Member	
Dr. Amjad Tuffaha Associate Professor Thesis Committee Member	
Dr. Hana Sulieman Head of Mathematics	
Dr. James Griffin CAS Graduate Programs Director	
Dr. Mahmoud Anabtawi Dean of CAS	
Dr. Khaled Assaleh Interim Vice Provost for Research and Graduate Studies	

"The true sign of intelligence is not knowledge but imagination."
"The difference between stupidity and genius is that genius has its limits."

Albert Einstein

Dedication

In loving memory of Dr. Ibrahim Sadek

Abstract

We reformulated the problem of training the neural networks model into a convex optimization problem by performing a local quadratic expansion of the cost function and adding the necessary constraints. We designed a new algorithm that extends the back propagation algorithm for parameters estimation by using second-order optimization methods. We computed the second order mixed partial derivatives of the cost function for a single hidden layer neural network model to construct the Hessian matrix. We used the Gauss-Newton approximation instead of the Hessian matrix to avoid the analytical computation of the second order derivative terms for higher order neural network topologies. To compare the accuracy and computational complexity of our proposed algorithm versus the standard back propagation we tested both algorithms in different applications, such as: regression, classification, and ranking.

Search Terms: Neural Networks, Convex Optimization, Gauss-Newton Matrix

Contents

List of Figures

Chapter 1

Introduction

In the field of learning from data, researchers aim to model an unknown relation between the features (input variables) and the labels (output variables). Supervised learning uses the training data (samples consist of paired data of features and labels measurement) to produce a function (model) that can predict new examples [1].

The quality of the measured data is a primary concern to take into account when attempting to learn from data. In some cases, we consider a fewer number of features than what we need due to practical reasons in data collection or due to the lack of understanding of the underlying physical phenomena. Also, the measured data can also contain noise, typically because of the measurement tool's internal structure and due to environmental conditions.

The mathematical models are either parametric or non-parametric models. A parametric model assumes a finite dimensional set of parameters. However, a non-parametric model doesn't hold the same assumption [8]. The complexity of parametric models increases as we increase the number of parameters [14]. In the case of Artificial Neural Networks (ANN) model complexity increases as we increase the number of layers and neurons.

Let $y \in \mathbb{R}^{m \times 1}$ be the labels vector (assuming single output) and $x \in \mathbb{R}^{m \times n}$ be the features matrix with m samples and n features. The mapping from x to y is unknown; instead the hypothesis $\hat{y_\theta}$ will approximate the unknown mapping. The parameters estimation of $\hat{y_\theta}$ is based on minimizing a cost function $C(\theta)$ w.r.t the parameters θ.

Function approximation is effectively made using methods based on analytical approaches such as series expansion and Stone-Weierstrass theorems [12]. Such

methods constrain the problem of learning from data by imposing the continuity of the unknown function. However, we do not know if the unknown function is continuous or if it exists.
In simple models like linear regression, parameters can be estimated optimally by computing a closed form equation that provides the optimal parameters $\hat{\theta}$. However, in more complex models like ANN parameters estimation is done in an iterative way since a closed form equation does not exist. For some applications, the computational time required to perform parameters estimation increases non-linearly as we increase the number of samples m. Let k be the number of samples needed to achieve parameters estimation. In ranking applications with pairwise training rule k is the square of m ($k = m^2$). Therefore, traditional algorithms like back propagation require additional time to estimate the parameters. Hence, new algorithms are required to compute the optimal parameters in a more efficient way.

Neural Networks

In chapter 2 of this thesis, we will formulate the ANN model analytically and state the back propagation algorithm which is the most widely used algorithm for training ANN [15]. ANN model maps a set of features x to a desired output y by spanning a composition (layers) of nonlinear activation functions acting on the span of features.
The performance of the model depends on our initial parameters, activation function, and the values of the learning rate η, momentum factor μ, and the batch size. η & μ define the step size in the parameters update loop and a decaying factor of the previous iterations gradient [17]. Both parameters improve the convergence of the back propagation algorithm. The comparison between the activation functions and their advantages/disadvantages is difficult, and such comparison will be biased by our choice of initial parameters [13]. Below are the most common choices of the activation function:

- Gaussian function

$$\phi(x_i) = \exp\left(-\frac{\|x_i - c_i\|_2^2}{2\sigma^2}\right)$$

- Multi-quadratic function

$$\phi(x_i) = \sqrt{\|x_i - c_i\|_2^2 + a^2}$$

- Soft-sign function

$$\phi(x_i) = \frac{x_i}{1 + |x_i|}$$

- Sigmoid function

$$\phi(x_i) = \sigma(x) = \frac{1}{1 + \exp(-x_i)}$$

The sigmoid function $\sigma(x)$ is the commonly used activation function in ANN . The main reasons behind choosing the sigmoid function are

- It has a bounded output range that varies from 0 to 1. Therefore, it is considered as a mapping of the input values into probabilities.
- The non-linearity induced in the network when using the sigmoid function enables the ANN to learn the nonlinear relation between the inputs and outputs.
- It is continuous and smooth which enables us to compute the derivatives which are needed when deriving the training rule for parameters estimation.

Another reason for justifying the choice of the sigmoid activation function is its natural probabilistic properties. Consider a two classes classification problem where each class follows a Gaussian distribution and both distributions have equal covariance matrices. We can show[13, 16] with Bayesian approach that the posterior probability of each class equals to the sigmoid activation function.

$$\begin{aligned} p(Class_1 \mid x) &= \frac{p(x \mid Class_1)p(Class_1)}{p(x \mid Class_1)p(Class_1) + p(x \mid Class_2)p(Class_2)} \\ &= 1 - p(Class_2 \mid x) = \sigma(y(x)) \end{aligned}$$

where $y(x)$ is

$$\ln\left(\frac{p(x \mid Class_1)P(Class_1)}{p(x \mid Class_2)P(Class_2)}\right)$$

With the same above reasoning, we can generalize the classification problem to handle multiple classes by introducing the softmax activation function instead of the sigmoid activation function.

The universal approximation theorem proven by Cybenko in 1989 [10] states that an ANN with a single hidden layer containing an N number of neurons can approximate continuous functions defined on compact subsets of $\mathbb{R}^n$.
In a simple example where we have only two features, an ANN model consisting of a single hidden layer with two neurons $\hat{y_\theta}$ will be modeled by:

$$\hat{y_\theta} = \sigma(\theta_{11}^2\sigma(\theta_{11}^1 x_1 + \theta_{21}^1 x_2 + \theta_{1b}^1) + \theta_{21}^2\sigma(\theta_{12}^1 x_1 + \theta_{22}^1 x_2 + \theta_{2b}^1) + \theta_{1b}^2)$$

Rewriting the above using $\sigma(x) = \dfrac{1}{1+\exp(-x)}$, we obtain

$$\hat{y_\theta} = \frac{1}{1+\exp(-\frac{\theta_{11}^2}{1+\exp(-\theta_{11}^1 x_1 - \theta_{21}^1 x_2 - \theta_{1b}^1)} - \frac{\theta_{21}^2}{1+\exp(-\theta_{12}^1 x_1 - \theta_{22}^1 x_2 - \theta_{2b}^1)} - \theta_{1b}^2)}$$

where the upper indexes represent the layer number and the lower indexes represent the neuron numbers from the $(i-1)^{th}$ layer to the i^{th} layer.
The following figure illustrate a neural network with a single hidden layer. In the figure, z represents the linear combination of the previous layer outputs weighted by the current layer parameters. The linear combination z is fed into the sigmoid activation function to have the current layer output a.

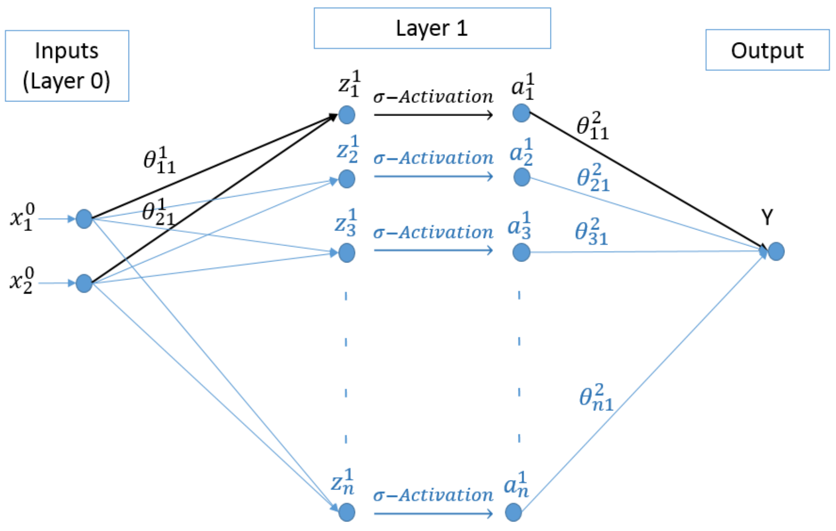


Figure 1.1: Single-Layer Artificial Neural Network

Convex Optimization

Convex optimization is the branch of mathematics that focuses on generalizing methods that can be used to minimize a convex objective function subjected to convex constraints. Convexity of the objective function can be geometrically interrupted as a function with a single unique minimum. A successful optimization of a convex problem guarantees to reach the global minimum of the objective function. On the other hand, optimization of a non-convex problem doesn't guarantee the convergence to the global minimum.
In chapter 3, Gradient Descent, Newton's, and Interior-Point Methods will be explained from the theory to implementation pseudo code. The choice of the optimization method depends on the problem's structure and complexity.
Gradient descent is a method used in unconstrained optimization problems and requires only the availability of the gradient ∇C of the objective function.
Newton's method is also used for unconstrained optimization. However, it requires the availability of the gradient ∇C and the Hessian matrix H of the objective function. Therefore, the objective function should be twice differentiable to use Newton's method.
The Interior-Point method is used in constrained optimization problems and requires both the gradient ∇C, and the Hessian matrix H. The interior-point method is used in large scale optimization problems where the number of parameters is large.
The term large scale optimization is vague and changes every year, since what is considered to be a large scale optimization problem a couple of years ago is now considered a regular problem due to the increase of the computational power. Optimizing neural networks with multiple layers, which are known as Deep Neural Networks (DNN), will require large scale optimization techniques since the number of parameters needed to be estimated can be considerably large w.r.t the available hardware.

Neural Networks as a Convex Problem

In this report, we will introduce a new way of training neural networks other than the traditional back propagation algorithm. After formulating the ANN model (Chapter 2) analytically and reviewing the general convex optimization methods

(Chapter 3), we will introduce training the ANN using convex optimization methods.
Training ANN using the back propagation algorithm is equivalent to an unconstrained minimization problem with a gradient descent algorithm. Since the ANN cost function is non-convex, ANN model training with the back propagation algorithm suffers the long training time and the convergence to local minimums instead of the global minimum.
By deriving the Jacobian and Hessian matrices of the ANN, we can implement faster algorithms like Newton's method. However, the non-convexity of the ANN will still limit the performance of the training since still there is no guarantee of converging to the global minimum
Finally, an approximated semi-definite matrix (Gauss-Newton Matrix) will be used to approximate the Hessian matrix. The approximate model introduced can be trained directly with convex optimization techniques. The addition of layers/neurons in the ANN will increase the complexity of training due to the large number of parameters need to be estimated. Therefore, we will formulate the problem using the interior point method after adding the necessary constraints. The interior point method is a second order large scale convex optimization technique. Also in this chapter, we will state the pseudo codes of our formulation with guidelines for implementation.
Simulation of the proposed approximate model will be presented within the concluding points. A lot of work can be done in the future to expand the approach to multi-layers ANN which is named as Deep Neural Networks (DNN). Also, the analytical derivations of the gradient and Hessian can be possibly rewritten in a closed form for a general DNN architecture. Such closed form formulas can increase the performance in numerical implementation.

Chapter 2

Neural Networks

ANN are systems designed based on the theory of the biological neuron system. With a similar architecture of the human brain neuron system researchers attempts to mimic the biological neuron system with an artificial system that can be used in many areas typically learning from data. ANN models can be used for classification, regression and ranking problems in a wide range of applications in engineering such as voice and image recognition, control theory..., etc. With more demand from engineering applications, the complexity of ANN has been increasing. For example, trying to analyze multiple layers of abstractions in an image or controlling an under-actuated robotic system requires more than a single hidden layer in the network.
As we increase the number of layers/neurons in the artificial neural network system, the network becomes a Deep Neural Network. With recent achievements training a deep neural network with a particular configuration is possible with an almost null error in the training set. However, training such networks is slow and complicated. Therefore, using deep architectures is limited to offline mode.

2.1 Forward Propagation

Let $\mathbf{M}$ be the number of hidden layer in the network. The first layer is called the input layer, and the last layer is called the output layer. Let $\mathbf{N}$ be the number of neurons in each hidden layer. Let l be a variable that denotes the layer number $(1 \leq l \leq \mathbf{M})$.

The following are the notations we will use in the ANN model:

$$\begin{cases} \theta^l & \text{Matrix of parameters from layer } l-1 \text{ to layer } l \\ \theta^l_b & \text{Vector of bias terms from layer } l-1 \text{ to layer } l \\ z^l & \text{Weighted input of layer } l \\ a^l & \text{Output of layer } l \end{cases}$$

Neural Networks maps the input features x into the hypothesis output $\hat{y_\theta}$ by applying a series of linear transformations (scaling with parameters θ) and nonlinear transformations (using the sigmoid function $\sigma(x)$). A process that is called forward propagation.
The linear output vector of layer l is the dot product between the layer l matrix of parameters θ^l and the previous layer output vector a^{l-1} plus the bias terms θ^l_b.

$$z^l = \theta^l \cdot a^{l-1^T} + \theta^l_b$$

Note that in forward propagation we are mapping the a^{l-1} to z^l with the dot product. Hence, we can stack a vector of l's to a^{l-1} to calculate z^1. Such matrix form of the operation improves the computation speed.

$$z^l = \begin{bmatrix} \theta^l_b & \theta^l \end{bmatrix} \cdot \begin{bmatrix} 1 \\ a^{l-1} \end{bmatrix}$$

Note that θ^l is the matrix of all the parameters that connects the outputs of the previous layer $l-1$ neurons with the current layer l neurons.
The linear output z^l is fed to a nonlinear activation function (sigmoid function) to reach the final output of the layer l by $a^l = \sigma(z^l)$. Similarly, the final output of the network is calculated as:

$$\begin{aligned} \hat{y_\theta} &= a^{M+1} = \sigma(z^{M+1}) \\ &= \sigma\big(\theta^{M+1} \cdot a^{M^T} + \theta^{M+1}_b\big) \end{aligned}$$

The features matrix x is considered as an input to the ANN and denoted as a^0. Using forward propagation a^0 propagates as explained earlier to compute all the hidden layers output a^l $(1 < l < \mathbf{M})$. The model output is considered to be the final layer output $\hat{y_\theta} = a^{M+1}$. The forward propagation can be summarized as the following:

Input Layer $l = 0$

$a^0 = x$ Features Matrix

First Layer $l = 1$

$z^1 = {a^0}^T \theta^1 + \theta_b^1$

$a^1 = \sigma(z^1)$

Second Layer $l = 2$

$z^2 = {a^1}^T \theta^2 + \theta_b^2$

$a^2 = \sigma(z^2)$

...

Output Layer $l = M + 1$

$z^{M+1} = {a^M}^T \theta^{M+1} + \theta_b^{M+1}$

$a^{M+1} = \sigma(z^{M+1})$

Output

$\hat{y_\theta} = \sigma(z^{M+1}) = a^{M+1}$

The following figure illustrates the multiple layer neural network system:

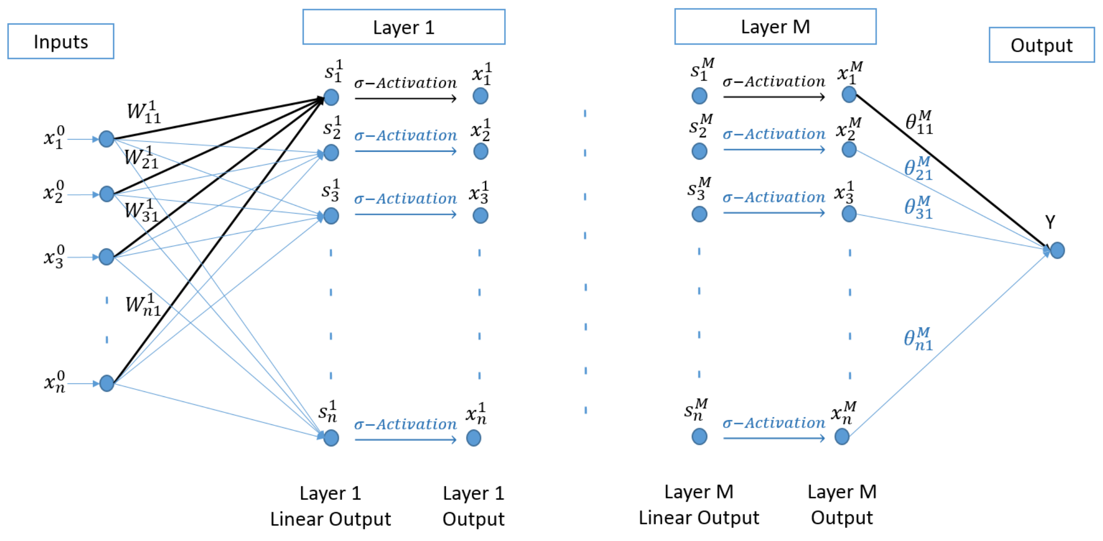


Figure 2.1: Multi-Layer Artificial Neural Network

2.2 Applications in Data Mining

In data mining, we are considering a large dataset with many features with an aim to extract useful information. Such information can be applied in many applications like environmental forecasts, stock market analysis, tumors identification..., etc. Artificial neural networks are considered as an adaptive model that can dynamically change according to the training set fed into it in the training phase. The adaptivity of the artificial neural networks system made it useful in many applications since it can be used in Regression, Classification, and with recent development, ANN can be used in Ranking problems also.

2.2.1 The Regression Problem

Regression analysis is a branch of mathematics that attempts to model the relationship between variables. Let $y = f(x_1, x_2, ..., x_n)$ be the output variable y modeled as a function of the features $x_1, x_2, ..., x_n$. The function $f : \mathbb{R}^n \to \mathbb{R}$ is unknown, and the regression analysis is applied to approximate the relation between the output variable y and the features. A typical application of regression is forecasting such as stock-market forecast, traffic flow forecast, and inventory forecast. Artificial neural networks can be viewed as a nonlinear regression model, also the more layers we add to the network the more levels of abstractions the model can learn.

$$f \approx \hat{y_\theta}$$

However with big data problems that include a high number of features, increasing the number of layers and neurons in the network becomes a must to meet the level of complexity of the problem. The complexity imposed by increasing the number of layers and neurons will highly impact the convexity of the cost function. For simple problems such as regression with two features and a single output, adding a layer to the network drives the cost function to have additional local minimums and hence parameters estimation becomes a non-convex problem. The complexity is highly correlated to our choice of the cost function. In regression, the standard cost function is the mean square error which is presented as:

$$\begin{aligned} C(y, \hat{y_\theta}) &= \frac{1}{2}(y - \hat{y_\theta})^2 \\ &= \frac{1}{2m}\sum_{i=1}^{m}(y_i - \hat{y_{\theta_i}})^2 \end{aligned}$$

Training such big networks or Deep Neural Networks is a hard problem and requires a high amount of computational power to estimate the optimal parameters of the model. Time series forecasting can also be viewed as a special type of regression although only a single feature is available to train the model. Recurrent Neural Networks (RNN) is a form of neural networks which have inner loops that make the model able to recall previous events learned. RNN models can determine the past trends in the time series data and compute predictions based on that [2].

2.2.2 The Classification Problem

Classification in machine learning is the problem that attempts to formulate a model that can be trained to classify a set of features into a particular category. Classification can be viewed as a discrete case of the regression problem since the number of outputs is always discrete. Let $y = f(x_1, x_2, ..., x_n)$ be the output set of classes. The function $f : \mathbb{R}^n \to \mathbb{N}$ is unknown and the classification problem aims to approximate it by the model $\hat{y_\theta}$. Artificial neural networks in binary classification (y is a set of only two classes) can be viewed as a generalization of the logistics regression. The Cross-Entropy cost function is defined as

$$\begin{aligned} C(y, \hat{y_\theta}) &= -y \ln \hat{y_\theta} - (1 - y)(1 - \ln \hat{y_\theta}) \\ &= \frac{-1}{m}\sum_{i}^{m}\Big(y_i \ln \hat{y_{\theta_i}} + (1 - y_i)(1 - \ln \hat{y_{\theta_i}})\Big) \end{aligned}$$

2.2.3 The Ranking Problem

The ranking problem corresponds to ranking a set of samples in order w.r.t relevance criteria. An important application to mention is the web page search engines such as Google, Yahoo, and Bing. As a machine learning definition, we refer to the problem as a preference learning on the pairwise sets of samples. We can judge the ranking of samples in different ways, for example, we can use the binary judgment set relevant, irrelevant or generalize it to multi-level ratings set great, good, fair, and bad. The choice of our judgment set always depends on the physical application. To illustrate the different approaches to this problem lets consider the following definition:

Definition 2.1. Let the relation $\triangleright$ imply "rank is relatively higher than" and $\triangleleft$ implies "rank is relatively smaller than". so we read $a_1 \triangleright a_2$ as a_1 has a rank relatively higher than a_2 where a_1, a_2 be any two samples.

Below are the different approaches used in the ranking problem:

- Pointwise Ranking
  The pointwise ranking assigns a unique value for each sample using a ranking function. Therefore, all the samples can be sorted based on their rank scores. Let $R : X \to \mathbb{R}$ be the pointwise ranking function such that $R(a_1) > R(a_2)$ if and only if $a_1 \triangleright a_2$. The pointwise ranking can be approached using regression algorithms to predict each sample ranking score approximately.

- Pairwise Ranking
  In this approach examine a pair of samples to see which sample has a higher rank relatively to the other. Let $R : X \times X \to \{-1, 0, 1\}$ be the pairwise ranking function such that:

$$R(a_1, a_2) = \begin{cases} -1 & \iff a_1 \triangleright a_2 \\ 0 & \iff a_1 = a_2 \\ 1 & \iff a_1 \triangleleft a_2 \end{cases}$$

  In pairwise ranking we are interested in finding a model that approximate the pairwise ranking function $R(a_1, a_2)$ based on the data provided in the training set. The main application of pairwise ranking is Web pages sorting.

- Listwise Ranking
  In this approach, we are assigning a relative rank score between two sets of samples. Let I be the set of ordered ranked-lists and $R : I \to \mathbb{R}$ be the listwise ranking function such that $R(i_1) > R(i_2)$ implies $i_1 \triangleright i_2$. Hence, $i_1, i_2 \in I$ implies $i_1 = (a_{i_{11}}, a_{i_{12}}, ..., a_{i_{1n}})$ and $i_2 = (a_{i_{21}}, a_{i_{22}}, ..., a_{i_{2n}})$. An application of the listwise approach is the structured ranking predictions.

In machine learning formulation the ranking problem can be approached as follows:

- Randomly select samples a_i from the training set.
- For each sample assign a relevance grade y_i.
- Model the ranking function $R(x)$ to preserve the order y.

In the ranking problem, there are many choices for the cost function. Below is the common two cost functions used:

- **Mean Average Precision (MAP)**
  The MAP requires a binary prejudgment relevant, irrelevant of each sample in the training data set. The precision at a point i is calculated as

  $$\text{Precision at i} = \frac{\text{number of relevant samples at top of i}}{i}$$

  The Average Precision (AP) is computed as

  $$AP = \sum_i \text{Precision at i} * \mathbf{I_i}$$

  where

  $$\mathbf{I_i} = \begin{cases} 1 \text{ if relevant} \\ 0 \text{ if irrelevant} \end{cases}$$

  The Mean Average Precision (MAP) is defined as the AP for the complete set

- **Normalized Discounted Cumulative Gain (NDCG)**
  NDCG at position k is calculated as

  $$NDCG_k = Z \sum_{i=1}^{k} \left(\frac{2^{R_i} - 1}{\log(i+1)} \right)$$

  Where

  $$\begin{cases} Z & \text{normalization factor} \\ R_i & \text{rank of sample } i \end{cases}$$

The pairwise ranking problem is not treated like regression and classification problem due to its complexity. Therefore, there exist some algorithms designed specifically to tackle the problem of ranking. Below is a survey of the standard algorithms used in parameters estimation for ranking applications.

- **PRanking Algorithm**
  PRanking is an ordinal regression algorithm [9] in which it attempts to find the optimal parameters θ that projects the samples into numeric scores which can be used to distinguish the ranks $r_1, r_2, ..., r_n$ of each sample. Hence we are assuming that there exist a rank $r_i \in \mathbb{R}$ for each sample. The algorithm updates the parameters by perceptron or SVM based algorithms.

$$R(x) = \begin{cases} 1 & \theta^T x < k_1 \\ i & k_{i-1} < \theta^T x < k_i \\ k & k_n < \theta^T x \end{cases}$$

- **Combined Regression and Ranking (CRR) Algorithm**
  The CRR algorithm uses a combined objective function that optimizes regression-based and rank-based objectives simultaneously. The combined CRR optimization problem is

$$\min_{\theta} \quad \alpha C(\theta, D) + (1-\alpha) C(\theta, P) + \frac{\lambda}{2} \|\theta\|_2^2$$

  Where C is the loss function, D is the training data set and P is a set of candidate pairs of samples. The CRR algorithm combines both the regression loss and the ranking loss function for the training and the parameters θ can be trained using gradient descent method.

- **RankNet Algorithm**
  RankNet algorithm is a probabilistic ranking model [5, 6] which predicts the probability of the relative rank $\hat{\mathrm{r}}_{ij}$ between a given pair of samples a_i, a_j. We define the modeled posterior as $p(a_i \rhd a_j)$ or $p(a_i \lhd a_j)$ where $a_i \rhd a_j$ asserts that sample a^i has a higher rank relatively to sample a_j.
  Let the training data consist of a pair of samples which shares the same features along with the target pairwise rank (output) $\overline{p}$ between the pair. Target probability for pairwise samples can be defined in many forms. In its

simplest forms, it is defined as:

$$\overline{p} = \begin{cases} 1 & a_i \rhd a_j \\ 0 & a_i, a_j \text{ has relatively the same rank} \\ -1 & a_i \lhd a_j \end{cases}$$

The RankNet algorithm models the pairwise relative rank probability using the neural networks framework by defining the estimated relative probability between two samples to be

$$\begin{aligned} \hat{r}_{ij} &\equiv \gamma(\hat{r}_i - \hat{r}_j) \\ p_{ij} &\equiv \frac{\exp(\hat{r}_{ij})}{1+\exp(\hat{r}_{ij})} = \sigma(\hat{r}_{ij}) \end{aligned} \tag{2.1}$$

Where $\hat{\mathrm{r}}_i, \hat{\mathrm{r}}_j$ are theoretically the outputs of an ANN model that assigns a pointwise rank to every sample. Although RankNet algorithm doesn't train the ANN to predict any pointwise ranks of the samples it uses the pointwise outcomes to construct a training rule that generalizes the back propagation algorithm to predict the pairwise relative rank $\hat{\mathrm{r}}_{ij}$ between any given two samples a_i, a_j.
Frank Algorithm is a generalization of the RankNet algorithm by using a new cost function. The fidelity cost function is defined as

$$F \equiv 1 - (\sqrt{\overline{p} * p} + \sqrt{(1-\overline{p}) * (1-p)})$$

The Frank algorithm provides a zero minimum cost within a bounded interval on $[0, 1]$ however the cost function is non-convex.

2.3 Back Propagation

Training the neural networks in all topologies follows the same principle. We start by defining our cost function C, and we derive the gradient ∇C with respect to all the parameters of the net. The back propagation algorithms utilize the theory of chain rule for computing the derivative of a composition of functions to derive the gradient of the cost function. Below is the derivation of the back propagation for the standard cost functions used in classification and regression applications. Note that deriving the gradient of the cost function for ranking applications is not straightforward [5], and therefore a generalization of the back propagation algorithm is needed.

$$C(y, \hat{y_\theta}) = \begin{cases} -y \ln \hat{y_\theta} - (1-y)(1 - \ln \hat{y_\theta}) & \text{Cross Entropy Loss} \\ \frac{1}{2}(y - \hat{y_\theta})^2 & \text{Mean Square Loss} \end{cases}$$

The back propagation algorithm computes the gradient of C with respect to all the parameters θ using the chain rule.

$$\frac{\partial C}{\partial \theta^{M+1}} = \frac{\partial C}{\partial \hat{y_\theta}} \frac{\partial \hat{y_\theta}}{\partial z^{M+1}} \frac{\partial z^{M+1}}{\partial \theta^{M+1}}$$

The three terms on the right hand side can be computed as the following:

- **Computing** $\frac{\partial C}{\partial \hat{y_\theta}}$
  The derivative of C with respect to $\hat{y_\theta}$ is straight forward, but it depends on our choice of C.

$$\frac{\partial C}{\partial \hat{y_\theta}} = \begin{cases} \frac{1-2y}{\hat{y_\theta}} & \text{Cross Entropy Loss} \\ y - \hat{y_\theta} & \text{Mean Square Loss} \end{cases}$$

- **Computing** $\frac{\partial \hat{y_\theta}}{\partial z^{M+1}}$
  By the definition of the ANN model $\hat{y_\theta} = \sigma(z^{M+1})$ hence we can directly compute the derivative as:

$$\frac{\partial \hat{y_\theta}}{\partial z^{M+1}} = \sigma'(z^{M+1})$$

- **Computing** $\frac{\partial z^{M+1}}{\partial \theta^{M+1}}$
  By the definition of the ANN model $z^{M+1} = \theta^{M+1} \cdot (a^M)^T + \theta_b^{M+1}$ so we can directly compute the derivative as:

$$\frac{\partial z^{M+1}}{\partial \theta^{M+1}} = a^M$$
$$\frac{\partial z^{M+1}}{\partial \theta_b^{M+1}} = 1$$

Finally by combining the above partial derivatives we can compute the gradient of the cost function C with respect to the final layer parameters θ^{M+1} and θ_b^{M+1}

(Assuming mean square loss).

$$\begin{aligned}\frac{\partial C}{\partial \theta^{M+1}} &= \big(y - \hat{y}_\theta\big)\sigma'(z^{M+1})a^M \\ \frac{\partial C}{\partial \theta_b^{M+1}} &= \big(y - \hat{y}_\theta\big)\sigma'(z^{M+1})\end{aligned}$$

We can simplify the common factor by introducing $\delta^{M+1} = \frac{\partial C}{\partial \hat{y}_\theta}\frac{\partial \hat{y}_\theta}{\partial z^{M+1}}$

$$\begin{aligned}\frac{\partial C}{\partial \theta^{M+1}} &= \delta^{M+1}a^M \\ \frac{\partial C}{\partial \theta_b^{M+1}} &= \delta^{M+1}\end{aligned}$$

We should repeat the steps above for the other layers parameters. For the parameters of the last hidden layer M

$$\begin{aligned}\frac{\partial C}{\partial \theta^{M}} &= \frac{\partial C}{\partial \hat{y}_\theta}\frac{\partial \hat{y}_\theta}{\partial z^{M+1}}\frac{\partial z^{M+1}}{\partial \theta^{M}} \\ &= \frac{\partial C}{\partial \hat{y}_\theta}\frac{\partial \hat{y}_\theta}{\partial z^{M+1}}\frac{\partial z^{M+1}}{\partial a^{M}}\frac{\partial a^{M}}{\partial \theta^{M}} \\ &= \delta^{M+1}\frac{\partial z^{M+1}}{\partial a^{M}}\frac{\partial a^{M}}{\partial \theta^{M}}\end{aligned}$$

The last two terms of the right hand side can be computed as:

- **Computing** $\frac{\partial z^{M+1}}{\partial a^M}$
  Recall that $z^{M+1} = \theta^{M+1}a^M + \theta_b^{M+1}$. By substitution:

$$\frac{\partial z^{M+1}}{\partial a^M} = \theta^{M+1}$$

- **Computing** $\frac{\partial a^{M}}{\partial \theta^M}$
  Recall that $a^M = \sigma(z^M)$ and $z^M = \theta^M a^{M-1} + \theta_b^M$. By substitution:

$$\begin{aligned}\frac{\partial a^M}{\partial \theta^M} &= \frac{\partial a^M}{\partial z^M}\frac{\partial z^M}{\partial \theta^M} = \sigma'(z^M)a^{M-1} \\ \frac{\partial a^M}{\partial \theta_b^M} &= \frac{\partial a^M}{\partial z^M}\frac{\partial z^M}{\partial \theta_b^M} = \sigma'(z^M)\end{aligned}$$

We can simplify again the common factor by introducing $\delta^M = \frac{\partial a^M}{\partial z^M} = \sigma'(z^M)$ to conclude the partial derivatives with respect layer M parameters as:

$$\frac{\partial C}{\partial \theta^M} = \delta^{M+1}\delta^M a^{M-1}$$
$$\frac{\partial C}{\partial \theta_b^M} = \delta^{M+1}\delta^M$$

We will continue to compute the gradient of the cost function C w.r.t all the other layer's parameters $M-1, M-2, ..., 1$. The first layer gradient will follow to be

$$\frac{\partial C}{\partial \theta^1} = \delta^{M+1}\delta^M \; ... \; \delta^2\delta^1 a^0$$
$$\frac{\partial C}{\partial \theta_b^1} = \delta^{M+1}\delta^M \; ... \; \delta^2\delta^1$$

The back propagation algorithm computes the gradient of the cost function ∇C with respect to all the parameters θ using the chain rule. The back propagation algorithm can be viewed in an algorithmic form [15] as shown in the following page:

```
1: Select the number of layers $M$. Set the value of the error tolerance parameter
   $\epsilon > 0$. Let $E$ be the batch training error.
2: Initialize the all the parameters in the matrix randomly $\theta \sim \mathcal{N}(0,1)$ for all
   layers.
3: Calculate the neural output.
       $z^l = {\theta^l}^T a^{l-1} + \theta_b^l$
       $a^l = \sigma(z^l)$
4: Calculate the output residue based on the choice of the cost function $C$.
       $\xi^{M+1} = \frac{\partial C}{\partial \hat{y_\theta}} = \begin{cases} \frac{1-2y}{\hat{y_\theta}} & \text{Cross Entropy Loss} \\ y - \hat{y_\theta} & \text{Mean Square Loss} \end{cases}$
5: **Repeat**
6: Calculate the last layer delta's.
       $\delta^{M+1} = \xi^{M+1}\sigma'(z^M)$
7: Recursively calculate the propagation errors of the hidden neurons.
       $\xi^l = \sum_{l=M}^{1} \delta^{l+1}\theta^l$
8: Recursively calculate the hidden neurons delta's vectors.
       $\delta^l = \xi^l \circ \sigma'(z^l)$
9: Return the partial derivative of the cost function with respect to each layer
   parameters.
       $\frac{\partial C^l}{\partial \theta} = \delta^l a^{l-1}$
       $\frac{\partial C^l}{\partial \theta_b} = \delta^l$
```

Algorithm 1: The Back Propagation Algorithm [15]

Chapter 3

Convex Optimization

Optimization methods are a set of algorithms that aim to minimize a convex objective function. The simplest geometric interpretation of the convexity of a cost function is the uniqueness of the minimum value (Convex functions has only a global minimum). Lets x_1 and x_2 be two points in the domain and t a parametrization variable then $\forall x_1, x_2 \in X, \forall t \in [0, 1]$ a function f is convex if and only if

$$f(tx_1 + (1-t)x_2) \leq tf(x_1) + (1-t)f(x_2)$$

By having a convex objective function defined on a convex set the optimization solution can reach the global minimum much faster than the non-convex case. Different algorithms will approach the global minimum solution point using different techniques and complexity. The first-order optimization methods like the gradient descent algorithm, only the gradient of the objective function is required to achieve the optimization solution.

$$\nabla f = \left(\frac{\partial f}{\partial x_1}, \cdots, \frac{\partial f}{\partial x_n}\right)$$

However, second-order optimization methods like Newton's, Trust Region, and Interior Points methods requires the availability of the gradient ∇C as well as the Hessian matrix H. A quick comparison between first-order and second-order optimization methods shows that first-order methods are faster and simpler. However, the second-order methods are more accurate but slower and have a higher complexity.

Depending on the complexity of the problem the computational time of the Hessian matrix is high. Also, in the second-order methods computing the inverse of the Hessian matrix is required which is not guaranteed to exists since the matrix might not be invertible especially for the non-convex cost functions. The Hessian matrix is computed generally as follows:

$$H = \begin{bmatrix} \dfrac{\partial^2 f}{\partial x_1^2} & \dfrac{\partial^2 f}{\partial x_1 \, \partial x_2} & \cdots & \dfrac{\partial^2 f}{\partial x_1 \, \partial x_n} \\ \dfrac{\partial^2 f}{\partial x_2 \, \partial x_1} & \dfrac{\partial^2 f}{\partial x_2^2} & \cdots & \dfrac{\partial^2 f}{\partial x_2 \, \partial x_n} \\ \vdots & \vdots & \ddots & \vdots \\ \dfrac{\partial^2 f}{\partial x_n \, \partial x_1} & \dfrac{\partial^2 f}{\partial x_n \, \partial x_2} & \cdots & \dfrac{\partial^2 f}{\partial x_n^2} \end{bmatrix}$$

Artificial neural networks are non-convex problems even if we choose a convex cost function (except neural networks with a single layer) since the number of layers will raise the level of complexity of the problem by having additional minimum values. As a result, the convexity feature of the problem is lost.

Reaching the global minimum value of the cost function C with the artificial neural networks model using convex optimization methods in an efficient time seems to be an invalid argument simply because the function is not convex.

In this thesis, we argue that the above problem can be treated as a convex optimization problem. We start by deriving the analytical form of the gradient and Hessian matrices of the cost function subjected to the artificial neural networks model. Later, we reformulate the cost function with a special approximation of the Hessian matrix. Finally, we use the approximated cost function as an objective function in our optimization problem formulation.

The approximated convex Hessian matrix will be derived using the Gauss-Newton matrix method. The approximated positive semi-definite Hessian matrix will be used in the second order expansion of the objective function. Optimizing the approximated convex objective function will be done using different optimization techniques.

The idea of using an approximated form of the Newton's method is called the inexact Newton's method [3]. In the inexact Newton's method, we attempt to solve an approximated Newton's system to derive only a good enough direction search for the minimum. In mathematical optimization form:

$$\min_{\theta} \quad C(y, \hat{y_\theta})$$

which will transforms into following equation (after expanding C)

$$\min_{\theta} \quad \theta^T \cdot \nabla C + \frac{1}{2}\theta^T \cdot \hat{\mathrm{H}}(\theta) \cdot \theta$$

In optimization theory, we are concerned with finding the value of θ which lies in a certain domain $\mathbf{D}$. The image $C(\theta)$ will have the minimum value in the range $\mathbf{R}$. Where θ is the set of parameters of our model.
The above minimization problem lies in the area of unconstrained optimization because there are no constraints. On the other hand, a constrained optimization problem will have the form of :

$$\begin{aligned} \min_{\theta} \quad & C(y, \hat{y_\theta}) \\ \text{subject to} \quad & f(\theta) \geq a \\ & g(\theta) \leq b \end{aligned}$$

Where $a, b \in \mathbb{R}$ and f, g are continuous functions on $\mathbb{R}$. Such problems are categorized under the area of constrained optimization problems. Many algorithms are used to find the parameters which minimize or maximize the objective function $C(\theta)$ such as Gradient Descent, Newton's Method..., etc.). More generally optimization problems can be interpreted as iterative methods to find the optimal values of the parameters w.r.t the problem constraints. A minimum or a maximum of any function occurs at the points at which the gradient vanishes.

$$\nabla C(\theta) = 0$$

Descent methods can be used to minimize the objective function C. However, for larger problems which contain more levels of abstractions, learning from data using the artificial neural networks model can be achieved by adding more layers and neurons in the network. In such cases, the classical descent methods used to minimize the objective function in an unconstrained manner will either not converge to the true global minimum and will require a high computational amount of time which is not feasible in the real-time applications.

The parameters estimation is done in an iterative way. At each iteration, the algorithm adds an additional value to the parameters of the previous iteration. The additional value to be added consists of two parts, a magnitude, and a direction. The magnitude value is controlled by the step size (also known as the learning rate), and the direction is a normalized vector quantity points in the direction of nearest local minimum (mathematically equal to the negative normalized gradient at each iteration).
Calculation of the gradient at each point of optimization can be achieved using algorithms developed precisely for training neural networks like the back propagation algorithm.
The general terminology of the descent methods is:

$$x = x + t_k \bigtriangleup x$$
$$\bigtriangleup x : \text{search direction}$$
$$t_k : \text{step size}$$

Pseudo-code

```
$x = x_0$(any initial value)
 Loop until stopping criterion $< \epsilon$
 $\bigtriangleup x$ = search direction
 $t_k$ = step size)
 $x = x + t_k \bigtriangleup x$
 Stopping criterion = depend on the method
```

3.1 Gradient Descent Method

We can see that the gradient descent method to derive an optimization rule is based on a greedy approach of search; we move in the direction opposite to the gradient of the function at any point.
This method is based on the idea that the gradient of the function is always in the direction of the maximum rate of change at that point.
The gradient descent method uses the following search direction:

$$\bigtriangleup x = -\nabla C$$

Pseudo-code

```
$x = x_0$(any initial value)
  Loop until stopping criterion $< \epsilon$
  $\triangle x = -\nabla C$
  $t_k =$ step size
  $x = x + t_k \triangle x$
  Stopping criterion $= \|\nabla C\|_2$
```

3.2 Newton's Method

Newton's method uses the search direction

$$\triangle x = -H(x)^{-1}\nabla f(x)$$

where $H(x)$ is the Hessian matrix. Newton's method in optimization is a generalization of Newton's method for finding the zeros of a function. Newton's method in optimization replaces the fraction $\frac{f(x^k)}{\nabla f(x^k)}$ by $\frac{\nabla f(x^k)}{\nabla^2 f(x^k)}$. In higher dimensions the division by $\nabla^2 f(x^k)$ becomes multiplication by the inverse of the matrix $\nabla^2 f(x^k)$. To derive the above using the Taylor expansion of the directional derivative of the function with the directional vector v.

$$\begin{aligned} \nabla f(x+v) &\sim f \\ f \ &= \nabla f(x) + \nabla^2 f(x)v \\ &= 0 \end{aligned}$$

Hence

$$\begin{aligned} v &= -\nabla^2 f(x)^{-1}\nabla f(x) \\ &= -H(x)^{-1}\nabla f(x) \\ &= \triangle x \end{aligned}$$

The final remark on Newton's method for optimization is that Newton's method uses the local Hessian near the point x and calculate the direction of minimization based on that local approximation.

Pseudo-code

```
$x = x_0$(any initial value)
  Loop until stopping criterion $< \epsilon$
  $\triangle x = -H(x)^{-1}\nabla C$
  $t_k =$ step size
  $x = x + t_k \triangle x$
  Stopping criterion $= \|\nabla C\|_2$
```

3.3 Interior Point Methods

So far we were discussing the traditional methods used in optimization such as gradient descent and Newton's methods. Such methods are mainly used for unconstrained optimization. However, most problems in practice have constraints. Let $f_i(\theta), g_j(\theta)$ for $i \leq n, j \leq m$ be the n number of inequality and m number of equality constraint functions.

$$\begin{aligned} \min_{\theta} \quad & f_0(\theta) \\ \text{subject to} \quad & f_i(\theta) \leq 0 \\ & g_j(\theta) = 0 \end{aligned} \tag{3.1}$$

and so there is a need to use more practical algorithms to optimize such problems. To optimize an objective function with respect to some constraints we use the theory of Lagrange multipliers. Let the Lagrangian of the problem be defined as:

$$L(\theta, \alpha, \beta) = f_0(\theta) + \sum_{i=1}^{n} \alpha_i f_i(\theta) + \sum_{j=1}^{m} \beta_j g_j(\theta)$$

To ensure that our constraints are satisfied we impose the following:

$$\max_{\alpha,\beta} \quad L(\theta, \alpha, \beta) = \begin{cases} f_0(\theta) & \text{, If conditions are satisfied} \\ \infty & \text{, Otherwise} \end{cases}$$

By defining the Lagrangian, we have transformed our problem from a constrained problem into unconstrained problem as:

$$\min_{\theta} \max_{\alpha,\beta} \quad L(\theta, \alpha, \beta)$$

However, in large scale optimization where the number of parameters to be estimated is huge, the Lagrangian will require the optimization algorithm to solve a n number of constraints by finding the gradient for each parameter and equate it to zero. Such method is not feasible when it come to tackling a large scale nonlinear system like Deep Neural Networks.
The interior point method approximates the original inequality constrained problem into a sequence of equality constrained problems using the log barrier function. Let μ be a relatively small positive scalar (often called the Barrier Parameter). By reformulating equation (3.1) into:

$$B(\theta, \mu) = -\mu \sum_{i=1}^{I} \log(-f_i(\theta))$$

the minimization problems is reformulated (Assuming $g_j(\theta) = 0$) to:

$$\min_{\theta} \quad \big(tf_0(\theta) + B(\theta, \mu)\big)$$

Note that the log barrier function is twice differentiable and convex.

$$\nabla B(\theta, \mu) = \nabla f_0(\theta) - \mu \sum_{i=1}^{m} \frac{1}{f_i(\theta)} \nabla f_i(\theta)$$

Pseudo-code

```
$x = x$(any initial strictly feasible value)
  $t_k = t_0$ (Such that $t_0 > 0$)
  Loop until stopping criterion $< \epsilon$
  Compute $x^\star$ by minimizing $\big(tf_0(\theta) + B(\theta, \mu)\big)$
  $t_k = t_k +$ any practical increment
```

We should note that minimizing the approximate log barrier function $tf_0(\theta) + B(\theta, \mu)$ is done with Newton's method. Hence the gradient and Hessian of the objective function are required. The solution of the interior point method is suboptimal to the original problem.

Chapter 4

Neural Networks as a Convex Problem

In this thesis, we aim to apply convex optimization techniques to ANN although the ANN is a non-convex problem. However, we are aiming to tackle the non-convexity of the problem with the second-order expansion of the cost function and by approximating the Hessian matrix using the Gauss-Newton matrix.
This approach of training ANN will be used to modify the RankNet algorithm that is used in ranking applications. Nevertheless, this approach can be used in other applications like regression and classification.
Training ANN with traditional methods leverage the back propagation algorithm to compute the gradient of the cost function ∇C with respect to the model parameters. Later on, the gradient is used to update the search direction in an attempt to find the global minimum of the cost function. Unfortunately, ANN has many local minimums and training ANN using back propagation does not guarantee a convergence to the global minimum.
Second-order optimization methods require computing the Hessian matrix. Although it is complicated and time-consuming, computing the Hessian matrix provides more efficiency in updating the search direction. Computing the Hessian matrix depends on the structure of the model.
In the case of artificial neural networks model, the large number of parameters makes it hard to compute the Hessian matrix analytically because we require computing the second order mixed partial derivatives with respect to all the parameters. In this thesis, a generalization of RankNet algorithm will be presented

by introducing both the Hessian matrix and the Gauss-Newton matrix in the training phase.
In the following sections, we will illustrate the idea of training ANN using convex optimization techniques, and we will show how to use such techniques in ranking theory, for simplicity, we will present the derivations and algorithms of a single layer network.

4.1 Neural Networks in Optimization Form

Artificial neural networks can be viewed as a minimization problem where the objective function is the cost function.

$$\min_{\theta} \quad C(\theta) \tag{4.1}$$

The above problem is typically approached using the gradient descent algorithm where the gradient is computed using the back propagation algorithm. However, since the cost function is non-convex, the gradient descent approach will not guarantee a convergence to the global minimum.

```
1: Initialize $\theta = \theta_{initial}$
2: Choose $\eta$ the learning rate.
3: Compute $\nabla C(\theta)$ using Back Propagation (Algorithm 1)
4: $\triangle\theta = -\nabla C$
5: $\theta = \theta + \eta \triangle \theta$
6: **Until** $C(\theta) < \epsilon$
```

Algorithm 2: Training ANN using the Back Propagation Algorithm

However the cost function $C(\theta)$ is a non-convex function so we will expand the cost function with Taylor expansion theory around the initial value of parameters $\theta_{initial}$. This approach preserves the local curvatures [7] around the initial values. Finally, we will apply convex optimization techniques to optimize the approximated convex cost function $\hat{C(\theta)}$.

$$\begin{aligned} \hat{C(\theta)} &= C(\theta + \delta\theta) \\ &= C(\theta) + \nabla C(\theta)^T \delta\theta + \frac{1}{2}\delta\theta^T H(\theta)\delta\theta \end{aligned} \tag{4.2}$$

Due to the complexity of deriving $H(\theta)$ analytically an approximated form of the Hessian will be used instead. The Gauss-Newton approximation [7] approximates the Hessian matrix locally around the initial parameters. It is calculated by computing the outer product of the gradient matrices.

$$\hat{H(\theta)} = \nabla C(\theta)^T \cdot \nabla C(\theta)$$

By replacing $H(\theta)$ by $\hat{H(\theta)}$ in equation (4.2) the quadratic approximation of equation (4.2) is called the Gauss-Newton approximation. Since the cost function (4.2) is the second order expansion of the cost function $C(\theta)$ we can use second order methods to minimize it. We can enhance the training of the ANN by approaching the problem with computing the search direction $\triangle\theta$ using Newton's method.

```
1: Initialize $\theta = \theta_{initial}$
2: Choose $\eta$ the learning rate.
3: Compute $\nabla C(\theta)$ using Back Propagation (Algorithm 1)
4: Compute $H(\theta)$
5: $\triangle\theta = -H_{Newton}(\theta)^{-1}\nabla C(\theta)$
6: $\theta = \theta + \eta \triangle \theta$
7: **Until** $C(\theta) < \epsilon$
```

Algorithm 3: Training ANN using Newton Method

Training ANN can still be achieve using Newton's like method with the Gauss-Newton matrix approximation.

```
1: Initialize $\theta = \theta_{initial}$
2: Choose $\eta$ the learning rate.
3: Compute $\nabla C(\theta)$ using Back Propagation (Algorithm 1)
4: $\hat{H(\theta)} = \nabla C(\theta)^T \nabla C(\theta)$
5: $\triangle\theta = -\hat{H(\theta)}^{-1} \nabla C(\theta)$
6: $\theta = \theta + \eta \triangle \theta$
7: **Until** $C(\theta) < \epsilon$
```

Algorithm 4: Training ANN using Gauss-Newton Matrix

In the following sections, we will show how training the ANN using algorithm 4 is superior to algorithm 1 in terms of time and accuracy.

The Gradient

The matrix of all the first order partial derivatives of the cost function C with respect to its parameters θ is:

$$\nabla C(\theta) = \begin{bmatrix} \frac{\partial C}{\partial \theta^{M+1}} & \frac{\partial C}{\partial \theta_b^{M+1}} & \dots & \frac{\partial C}{\partial \theta^1} & \frac{\partial C}{\partial \theta_b^1} \end{bmatrix}$$

Using the back propagation algorithm we could analytically derive the partial derivatives as:

$$\nabla C(\theta) = \begin{bmatrix} \delta^M a^{M-1} & \delta^M & \dots & \delta^1 a^0 & \delta^1 \end{bmatrix}$$

For a single hidden layer with N number of neurons the gradient ∇C is:

$$\nabla C(\theta) = \begin{bmatrix} \frac{\partial C}{\partial \hat{y}}\sigma'(z^2)a^1 & \frac{\partial C}{\partial \hat{y}}\sigma'(z^2) & \theta^2\sigma'(z^1)a^0 & \theta^2\sigma'(z^1) \end{bmatrix}$$

The Hessian Matrix

The Hessian matrix is the matrix of all second order mixed partial derivatives. For simplicity, we will consider the single hidden layer neural networks model. The partial derivatives will be computed with respect to four parameters $\theta^2, \theta_b^2, \theta^1, \theta_b^1$ hence the Hessian matrix will consist of the following four columns:

$$H^1 = \begin{bmatrix} \frac{\partial^2 C}{\partial \theta^2 \partial \theta^2} \\ \frac{\partial^2 C}{\partial \theta_b^2 \partial \theta^2} \\ \frac{\partial^2 C}{\partial \theta^1 \partial \theta^2} \\ \frac{\partial^2 C}{\partial \theta_b^1 \partial \theta^2} \end{bmatrix} \quad H^2 = \begin{bmatrix} \frac{\partial^2 C}{\partial \theta^2 \partial \theta_b^2} \\ \frac{\partial^2 C}{\partial \theta_b^2 \partial \theta_b^2} \\ \frac{\partial^2 C}{\partial \theta^1 \partial \theta_b^2} \\ \frac{\partial^2 C}{\partial \theta_b^1 \partial \theta_b^2} \end{bmatrix} H^3 = \begin{bmatrix} \frac{\partial^2 C}{\partial \theta^2 \partial \theta^1} \\ \frac{\partial^2 C}{\partial \theta_b^2 \partial \theta^1} \\ \frac{\partial^2 C}{\partial \theta^1 \partial \theta^1} \\ \frac{\partial^2 C}{\partial \theta_b^1 \partial \theta^1} \end{bmatrix} \quad H^4 = \begin{bmatrix} \frac{\partial^2 C}{\partial \theta^2 \partial \theta_b^1} \\ \frac{\partial^2 C}{\partial \theta_b^2 \partial \theta_b^1} \\ \frac{\partial^2 C}{\partial \theta^1 \partial \theta_b^1} \\ \frac{\partial^2 C}{\partial \theta_b^1 \partial \theta_b^1} \end{bmatrix} \tag{4.3}$$

The detailed derivations can be found in the appendix section. The columns of the Hessian matrix for a single hidden layer network is:

$$H^1 = \begin{bmatrix} a^1(\frac{\partial C}{\partial \hat{y}}\sigma''(z^2){a^1}^T + \sigma'(z^2)\frac{\partial \frac{\partial C}{\partial \hat{y}}}{\partial \theta^2}) \\ a^1(\frac{\partial C}{\partial \hat{y}}\sigma''(z^2) + \sigma'(z^2)\frac{\partial \frac{\partial C}{\partial \hat{y}}}{\partial \theta_b^2}) \\ \frac{\partial C}{\partial \hat{y}}\sigma'(z^2)\sigma'(z^1){a^0}^T + a^1(\frac{\partial C}{\partial \hat{y}}\sigma''(z^2){a^0}^T\theta^2 + \sigma'(z^2)\frac{\partial \frac{\partial C}{\partial \hat{y}}}{\partial \theta^1}) \\ \frac{\partial C}{\partial \hat{y}}\sigma'(z^2)\sigma'(z^1) + a^1(\frac{\partial C}{\partial \hat{y}}\sigma''(z^2)\theta^2 + \sigma'(z^2)\frac{\partial \frac{\partial C}{\partial \hat{y}}}{\partial \theta_b^1}) \end{bmatrix}$$

$$H^2 = \begin{bmatrix} \frac{\partial C}{\partial \hat{y}} \sigma''(z^2) {a^1}^T + \sigma'(z^2) \frac{\partial \frac{\partial C}{\partial \hat{y}}}{\partial \theta^2} \\ \frac{\partial C}{\partial \hat{y}} \sigma''(z^2) + \sigma'(z^2) \frac{\partial \frac{\partial C}{\partial \hat{y}}}{\partial \theta^2_b} \\ \frac{\partial C}{\partial \hat{y}} \sigma''(z^2) {a^0}^T \theta^2 + \sigma'(z^2) \frac{\partial \frac{\partial C}{\partial \hat{y}}}{\partial \theta^1} \\ \frac{\partial C}{\partial \hat{y}} \sigma''(z^2) \theta^2 + \sigma'(z^2) \frac{\partial \frac{\partial C}{\partial \hat{y}}}{\partial \theta^1_b} \end{bmatrix}$$

$$H^3 = \begin{bmatrix} \sigma'(z^1) a^0 \\ 0 \\ \theta^2 \sigma''(z^1) {a^0}^T a^0 \\ \theta^2 \sigma''(z^1) a^0 \end{bmatrix}$$

$$H^4 = \begin{bmatrix} \sigma'(z^1) \\ 0 \\ \theta^2 \sigma''(z^1) {a^0}^T \\ \theta^2 \sigma''(z^1) \end{bmatrix}$$

We still need to compute the partial derivatives of the cost function with respect to all the parameters. Recall that the partial derivative of the cost function with respect to the artificial neural networks model output $\hat{y}_\theta$ is:

$$\frac{\partial C}{\partial \hat{y}_\theta} = \begin{cases} \frac{1-2y}{\hat{y}_\theta} & \text{Cross Entropy Loss} \\ y - \hat{y}_\theta & \text{Mean Square Loss} \end{cases}$$

And writing $\hat{y}_\theta$ explicitly as a function of all the parameters of the network it follows that:

$$\begin{aligned} \hat{y}_\theta &= a^2 \\ &= \sigma(z^2) \\ &= \sigma({a^1}^T \theta^2 + \theta^2_b) \\ &= \sigma(\sigma(z^1)^T \theta^2 + \theta^2_b) \\ &= \sigma(\sigma({a^0}^T \theta^1 + \theta^1_b)^T \theta^2 + \theta^2_b) \end{aligned}$$

Taking the above analysis for the case of the Cross Entropy cost function:

$$\frac{\partial \frac{\partial C}{\partial \hat{y}_\theta}}{\partial \theta^2} = \left(\frac{1-2y}{\hat{y}_\theta}\right)\sigma'(z^2)a^1\theta^2 {a^0}^T$$
$$\frac{\partial \frac{\partial C}{\partial \hat{y}_\theta}}{\partial \theta_b^2} = \left(\frac{1-2y}{\hat{y}_\theta}\right)\sigma'(z^2)a^1\theta^2$$
$$\frac{\partial \frac{\partial C}{\partial \hat{y}_\theta}}{\partial \theta^1} = \left(\frac{1-2y}{\hat{y}_\theta}\right)\sigma'(z^2)a^1$$
$$\frac{\partial \frac{\partial C}{\partial \hat{y}_\theta}}{\partial \theta_b^1} = \left(\frac{1-2y}{\hat{y}_\theta}\right)\sigma'(z^2)$$

Taking the above analysis for the case of the Mean Square cost function:

$$\frac{\partial \frac{\partial C}{\partial \hat{y}_\theta}}{\partial \theta^2} = -\sigma'(z^2)a^1\theta^2 {a^0}^T$$
$$\frac{\partial \frac{\partial C}{\partial \hat{y}_\theta}}{\partial \theta_b^2} = -\sigma'(z^2)a^1\theta^2$$
$$\frac{\partial \frac{\partial C}{\partial \hat{y}_\theta}}{\partial \theta^1} = -\sigma'(z^2)a^1$$
$$\frac{\partial \frac{\partial C}{\partial \hat{y}_\theta}}{\partial \theta_b^1} = -\sigma'(z^2)$$

Combining the Hessian matrix columns H^1, H^2 with the above equations depending on the choice of the cost function analytical derivation of the Hessian matrix of a single hidden layer network is completed. The Gauss-Newton approximation of the Hessian matrix [7] is:

$$\hat{H(\theta)} = \nabla C(\theta)^T \cdot \nabla C(\theta) = \begin{bmatrix} \delta^2 a^1 \delta^2 a^1 & \delta^2 a^1 \delta^2 & \delta^2 a^1 \delta^1 a^0 & \delta^2 a^1 \delta^1 \\ \delta^2 a^1 \delta^2 & \delta^2 \delta^2 & \delta^2 \delta^1 a^0 & \delta^2 \delta^1 \\ \delta^1 a^0 \delta^2 a^1 & \delta^1 a^0 \delta^2 & \delta^1 a^0 \delta^1 a^0 & \delta^1 a^0 \delta^1 \\ \delta^2 a^1 \delta^1 & \delta^2 \delta^1 & \delta^1 a^0 \delta^1 & \delta^1 \delta^1 \end{bmatrix} \tag{4.4}$$

Unfortunately, it 's hard to obtain the exact form of the Hessian matrix analytically if the number of layers is more than one. To resolve this problem, we will use the Gauss-Newton approximation instead.

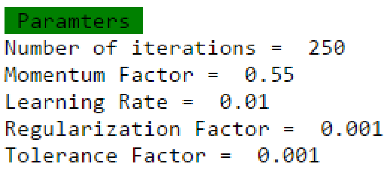
4.2 Training with Unconstrained Optimization

4.2.1 Regression

Friedman data set one [4] is used to simulate a response with a uniformly distributed parameters over $[0, 1]$

$$y = 10\sin(\pi x_1 x_2) + 20(x_3 - 0.5)^2 + 10x_4 + 5x_5 + \epsilon$$

The network was initiated with the following parameters:



```
Paramters
Number of iterations =  250
Momentum Factor =  0.55
Learning Rate =  0.01
Regularization Factor =  0.001
Tolerance Factor =  0.001
```

FIGURE 4.1: Cost Function for Friedman Data Set

Comparing the cost function reduction over the training iterations for both the traditional back propagation algorithm and the proposed algorithm (Gauss-Newton) is illustrated in the following figure:

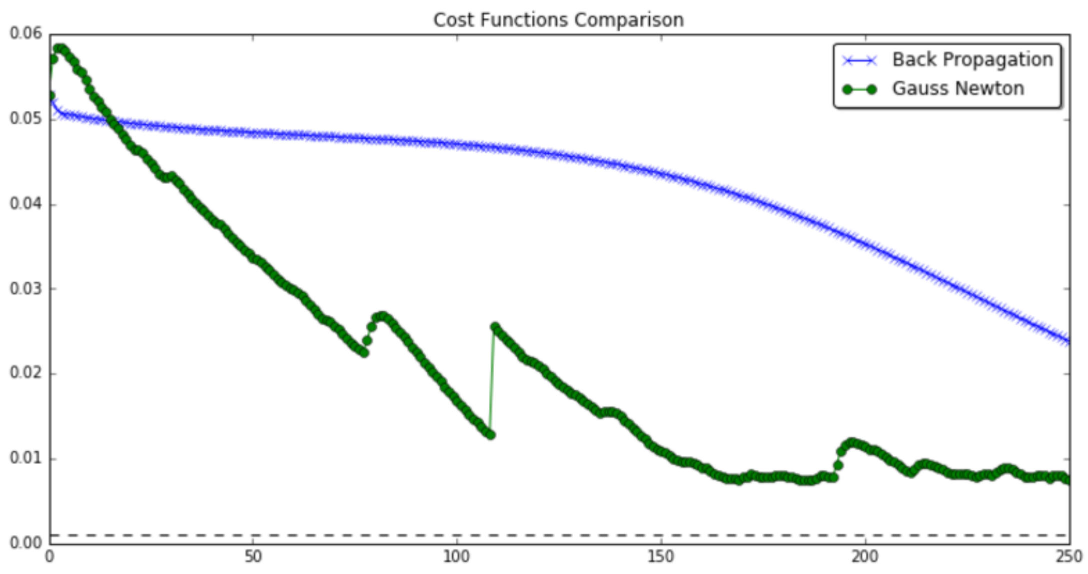
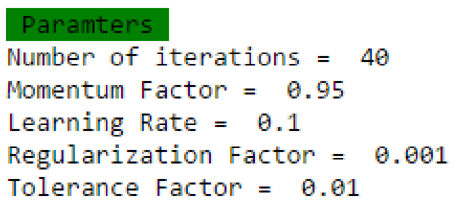


FIGURE 4.2: Cost Function for Friedman Data Set

For a bigger data set testing the Abalone data set is used. It consist of eight features with 41493 samples. The network were initiated with the following parameters: Comparing the cost function reduction over the training iterations for both



```
Paramters
Number of iterations =  40
Momentum Factor =  0.95
Learning Rate =  0.1
Regularization Factor =  0.001
Tolerance Factor =  0.01
```

FIGURE 4.3: Parameters for Abalone Data Set

the traditional back propagation algorithm and the proposed algorithm (Gauss-Newton) is illustrated in the following figure:

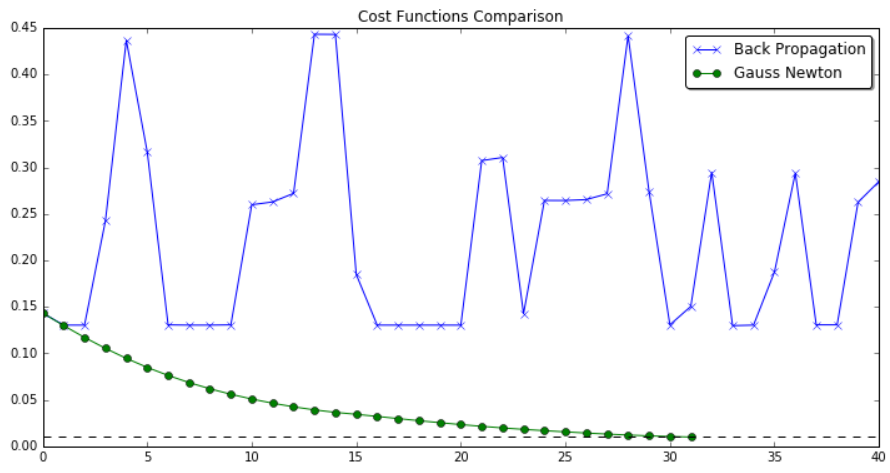
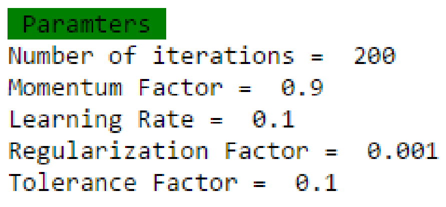


FIGURE 4.4: Cost Function for Abalone Data Set

4.2.2 Classification

Taking a small data set that presents a two classes output with two features and 400 samples. By training the network with the below parameters:



```
Paramters
Number of iterations =  200
Momentum Factor =  0.9
Learning Rate =  0.1
Regularization Factor =  0.001
Tolerance Factor =  0.1
```

FIGURE 4.5: Parameters for Moons Data Set

Comparing the cost function reduction over the training iterations for both the traditional back propagation algorithm and the proposed algorithm (Gauss-Newton) is illustrated in the following figure: For a bigger data set, we will test the model

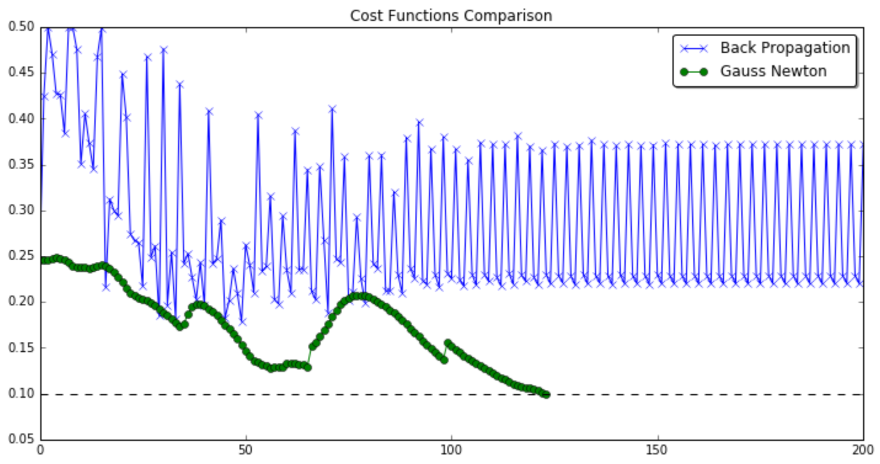
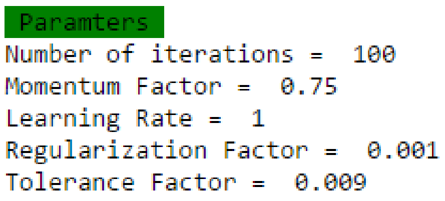


FIGURE 4.6: Cost Function for Moons Data Set

with the HIVA (existence of HIV disease in the human body data set) data set. It consist of 1618 features with 3849 samples. By training the network with the below parameters:



```
Paramters
Number of iterations =  100
Momentum Factor =  0.75
Learning Rate =  1
Regularization Factor =  0.001
Tolerance Factor =  0.009
```

FIGURE 4.7: Parameters for HIVA Data Set

Comparing the cost function reduction over the training iterations for both the traditional back propagation algorithm and the proposed algorithm (Gauss-Newton) is illustrated in the following figure:

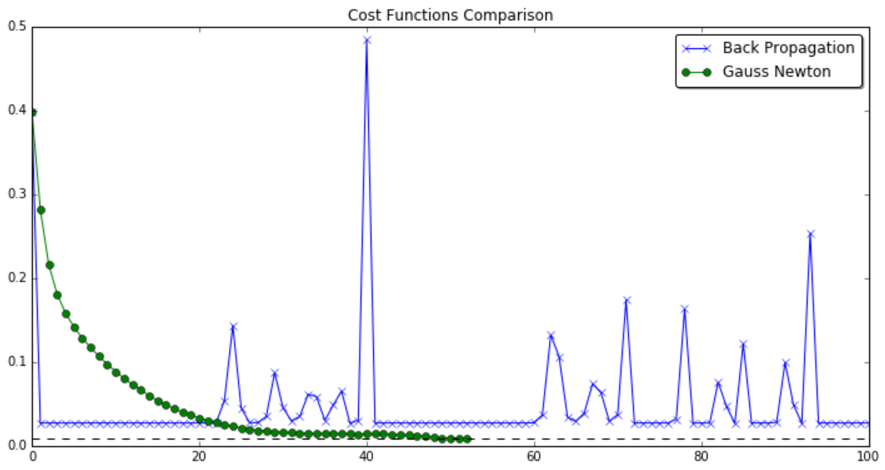


FIGURE 4.8: Cost Function for HIVA Data Set

4.2.3 Ranking

Let $\hat{\text{r}} : \mathbb{R}^d \to \mathbb{R}$ be our hypothesis to predict the pointwise rank $\hat{r}_i$ of the sample a_i. Let $\hat{r}_{ij}$ be the predicted relative rank between the two samples a_i, a_j and let r_{ij} be the target relative rank between . Define $\hat{r}_{ij} \equiv \gamma(\hat{r}_i - \hat{r}_j)$ where $\gamma \in [0, 1]$. In pair-wise ranking we are interested in the probability of relative rank between given two samples. Therefore, RankNet algorithm modifies the traditional back propagation algorithm to handle a pair of inputs instead of a single input. The RankNet modification [6] for a single layer neural network:

$$\begin{aligned}
\frac{\partial C}{\partial \theta^2} &= \delta_2^2 a_2^1 - \delta_1^2 a_1^1 \\
\frac{\partial C}{\partial \theta_b^2} &= \delta_2^2 - \delta_1^2 \\
\frac{\partial C}{\partial \theta^1} &= \delta_2^1 a_2^0 - \delta_1^1 a_1^0 \\
\frac{\partial C}{\partial \theta_b^1} &= \delta_2^1 - \delta_1^1
\end{aligned}$$

The cross entropy cost function will computes the error between the relative rank probability output p_{ij} and the target relative rank $\overline{p_{ij}}$

$$C(\overline{p_{ij}}, p_{ij}) = \overline{p_{ij}} \ln p_{ij} + (1 - \overline{p_{ij}})(1 - \ln p_{ij})$$

To derive the δ of the output layer for both samples we will require to compute the partial derivative of the cost function C with respect to the parameters θ.

$$\frac{\partial C}{\partial \theta} = \frac{\partial C}{\partial p_{ij}} \frac{\partial p_{ij}}{\partial \hat{r_{ij}}} = \frac{\partial C}{\partial \hat{r}_i} \frac{\partial \hat{r}_i}{\partial \theta} - \frac{\partial C}{\partial \hat{r}_j} \frac{\partial \hat{r}_j}{\partial \theta} \tag{4.5}$$

The problem will be arranged such that the sample i will always have a higher rank relative to j. Hence $\overline{p_{ij}} \equiv 1$ and with the necessary arrangement [6] the partial derivatives of the pointwise ranks can be linked as

$$\frac{\partial \hat{r}_i}{\partial \theta} = -\frac{\partial \hat{r}_j}{\partial \theta}$$

The following algorithm explains the details of implementing the RankNet algorithm: Second order optimization methods require the computation of a Newton step which requires computing the Hessian matrix or the Gauss-Newton matrix. However, we are expanding the cost function using the Gauss-Newton matrix instead of the Hessian matrix.

```
1: Initialize $\theta = \theta_{initial}$ randomly for all the network parameters.
2: Calculate the neural output for both samples $i$ and $j$.
       $\hat{r}_i = a_i^1 = \sigma(z^1)$
       $\hat{r}_j = a_j^1 = \sigma(z^1)$
3: Repeat
4:     Compute $\hat{\mathrm{r}}_{ij}$, $p_{ij}$ and $\pi_{ij}$
           $\hat{r}_{ij} \equiv \gamma(\hat{r}_i - \hat{r}_j)$
           $p_{ij} \equiv \frac{\exp(\hat{r}_{ij})}{1+\exp(\hat{r}_{ij})}$
           $\pi_{ij} = \gamma(p_{ij} - \overline{p}_{ij})$
5:     Compute
           $\xi_i^2 = \frac{\partial C}{\partial \hat{\mathrm{r}}_i} = \pi_{ij} \quad , \xi_j^2 = \frac{\partial C}{\partial \hat{\mathrm{r}}_j} = -\pi_{ij}$
6:     Calculate the last layer delta's for both samples $i$ and $j$.
           $\delta_i^2 = \xi_i^2 \sigma'(z^2) \quad , \delta_j^2 = \xi_j^2 \sigma'(z^2)$
7:     Recursively calculate the propagation errors of the hidden neurons.
           $\xi_i^1 = \delta_i^2 \theta^1 \quad , \xi_j^1 = \delta_j^2 \theta^1$
8:     Recursively calculate the hidden neurons delta's vectors.
           $\delta_i^1 = \xi_i^1 \circ \sigma'(z_i^1) \quad , \delta_j^1 = \xi_j^1 \circ \sigma'(z_j^1)$
9:     Update the parameters.
           $\begin{bmatrix} \theta^2 \\ \theta_b^2 \\ \theta^1 \\ \theta_b^1 \end{bmatrix} = \begin{bmatrix} \theta^2 \\ \theta_b^2 \\ \theta^1 \\ \theta_b^1 \end{bmatrix} + \eta \begin{bmatrix} \delta_i^2 a_i^1 - \delta_j^2 a_j^1 \\ \delta_i^2 - \delta_j^2 \\ \delta_i^1 a_i^0 - \delta_j^1 a_j^0 \\ \delta_i^1 - \delta_j^1 \end{bmatrix}$
10:    Compute the forward propagation with the updated parameters
11:    Compute the posterior probability $p_{ij}$
12: Until $E < \epsilon$
       $E = \frac{1}{m} \sum^{m} -\overline{p}_{ij} \log p_{ij} - (1 - \overline{p}_{ij}) \log(1 - p_{ij})$
```

Algorithm 5: The Back Propagation Algorithm Modified by RankNet

Deriving the second order derivatives of the cost function C with respect to the parameters θ can be done by directly by differentiating the gradient as computed by the RankNet (modified back propagation) with respect to θ.

$$\begin{aligned}
\frac{\partial}{\partial\theta}\left(\frac{\partial C}{\partial\theta}\right) = H_{ij} &= \frac{\partial}{\partial\theta}\left(\pi_{ij}\left(\frac{\partial\hat{r}_i}{\partial\theta} - \frac{\partial\hat{r}_j}{\partial\theta}\right)\right) \\
&= \frac{\partial\pi_{ij}}{\partial\theta}\frac{\partial C}{\partial\theta} + \pi_{ij}\left(\frac{\partial^2\hat{r}_i}{\partial\theta^2} - \frac{\partial^2\hat{r}_j}{\partial\theta^2}\right) \\
&= -\gamma\frac{\partial}{\partial r_{ij}}\left(\frac{1}{1+e^{r_{ij}}}\right)\frac{\partial r_{ij}}{\partial\theta}\frac{\partial C}{\partial\theta} + \pi_{ij}\left(\frac{\partial^2\hat{r}_i}{\partial\theta^2} - \frac{\partial^2\hat{r}_j}{\partial\theta^2}\right) \\
&= \gamma\gamma\frac{\partial}{\partial r_{ij}}\left(\frac{1}{1+e^{r_{ij}}}\right)\frac{\partial C}{\partial\theta}^T\frac{\partial C}{\partial\theta} + \pi_{ij}\left(\frac{\partial^2\hat{r}_i}{\partial\theta^2} - \frac{\partial^2\hat{r}_j}{\partial\theta^2}\right) \\
&= \gamma\gamma\left(\frac{-}{1+e^{r_{ij}}}\frac{1}{1+e^{-r_{ij}}}\right)\frac{\partial C}{\partial\theta}^T\frac{\partial C}{\partial\theta} + \pi_{ij}\left(\frac{\partial^2\hat{r}_i}{\partial\theta^2} - \frac{\partial^2\hat{r}_j}{\partial\theta^2}\right) \\
&= \gamma\pi p_{ij}\frac{\partial C}{\partial\theta}^T\frac{\partial C}{\partial\theta} + \pi_{ij}\left(\frac{\partial^2\hat{r}_i}{\partial\theta^2} - \frac{\partial^2\hat{r}_j}{\partial\theta^2}\right)
\end{aligned} \tag{4.6}$$

Note that the above equation can be approximated using Gauss-Newton matrix.

$$H_{ij} = \gamma\pi p_{ij}\frac{\partial C}{\partial\theta}^T\frac{\partial C}{\partial\theta} + \pi_{ij}\big(H_i - H_j\big)$$

Where H_i, H_j are the Hessian matrices of the neural networks constructed to model $\hat{r}_i$ and $\hat{r}_j$. We can read the above equation as the Hessian of the cost function equal to a factor times the outer product of the cost function gradient plus a factor times the difference of the pointwise model outputs Hessian matrices. Gauss-Newton approximation can approximate both Hessian matrices (H_i, H_j) and the result is used to train the ranking problem using Newton's methods.

$$\begin{aligned}
\hat{H}_{ij} &= \gamma\pi p_{ij}\frac{\partial C}{\partial\theta}^T\frac{\partial C}{\partial\theta} + \pi_{ij}\left(\frac{\partial\hat{r}_i}{\partial\theta}^T\frac{\partial\hat{r}_i}{\partial\theta} - \frac{\partial\hat{r}_j}{\partial\theta}^T\frac{\partial\hat{r}_j}{\partial\theta}\right) \\
&= \gamma\pi p_{ij}\frac{\partial C}{\partial\theta}^T\frac{\partial C}{\partial\theta} + \pi_{ij}\big(\hat{H}_i - \hat{H}_j\big)
\end{aligned} \tag{4.7}$$

Now we can illustrate how to enhance the RankNet algorithm by utilizing Newton's method to train the artificial neural networks model. The Gauss-Newton approximation combined with RankNet algorithm is listed as a new algorithm in the following page:

```
1: Initialize $\theta = \theta_{initial}$ randomly for all the network parameters.
2: Calculate the neural output for both samples $\hat{\mathrm{r}}_i = a_i^1 = \sigma(z^1)$ and
   $\hat{\mathrm{r}}_j = a_j^1 = \sigma(z^1)$.
3: **Repeat**
4: Compute $\hat{\mathrm{r}}_{ij}$, $p_{ij}$ and $\pi_{ij}$
   $$\hat{r}_{ij} \equiv \gamma(\hat{r}_i - \hat{r}_j)$$
   $$p_{ij} \equiv \frac{\exp(\hat{r}_{ij})}{1+\exp(\hat{r}_{ij})}$$
   $$\pi_{ij} = \gamma(p_{ij} - \overline{p}_{ij})$$
5: Compute $\xi_i^2 = \frac{\partial C}{\partial \hat{\mathrm{r}}_i} = \pi_{ij}$, $\xi_j^2 = \frac{\partial C}{\partial \hat{\mathrm{r}}_j} = -\pi_{ij}$
6: Calculate the last layer delta's for both samples $\delta_i^2 = \xi_i^2 \sigma'(z^2)$ and
   $\delta_j^2 = \xi_j^2 \sigma'(z^2)$.
7: Recursively calculate the propagation errors of the hidden neurons $\xi_i^1 = \delta_i^2 \theta^1$,
   $\xi_j^1 = \delta_j^2 \theta^1$.
8: Recursively calculate the hidden neurons delta's vectors $\delta_i^1 = \xi_i^1 \circ \sigma'(z_i^1)$,
   $\delta_j^1 = \xi_j^1 \circ \sigma'(z_j^1)$.
9: Compute the gradient of the cost function
   $$\nabla C = \begin{bmatrix} \delta_i^2 a_i^1 - \delta_j^2 a_j^1 \\ \delta_i^2 - \delta_j^2 \\ \delta_i^1 a_i^0 - \delta_j^1 a_j^0 \\ \delta_i^1 - \delta_j^1 \end{bmatrix}$$
10: Compute the Gauss-Newton matrices for both samples outputs $\hat{H}_i, \hat{H}_j$
11: Compute the approximated Hessian matrix for the cost function.
   $$\hat{H}_{ij} = \gamma \pi_{ij} p_{ij} \nabla C^T \cdot \nabla C + \pi_{ij}(\hat{H}_i - \hat{H}_j)$$
12: Update the parameters.
   $$\begin{bmatrix} \theta^2 \\ \theta_b^2 \\ \theta^1 \\ \theta_b^1 \end{bmatrix} = \begin{bmatrix} \theta^2 \\ \theta_b^2 \\ \theta^1 \\ \theta_b^1 \end{bmatrix} + \eta(\hat{H}_{ij}^{-1} \nabla C)$$
13: Compute the forward propagation with the updated parameters
14: Compute the posterior probability $p_{ij}$
15: **Until** $E < \epsilon$
   $$E = \frac{1}{m} \sum^{m} -\overline{p}_{ij} \log p_{ij} - (1 - \overline{p}_{ij}) \log(1 - p_{ij})$$
```

Algorithm 6: The Generalization of RankNet Algorithm Using Gauss-Newton Matrix

To test the proposed algorithm, five data sets were chosen, MQ2007, MQ2009 (benchmarked datasets designed especially for learning how to rank application), HIVA (existence of HIV disease in the human body dataset), SYLVA (ecology dataset) and NOVA (test classification dataset). For each data set the parameters used and the cost function reduction on the training data is illustrated in the following figures:
Data set: MQ2008

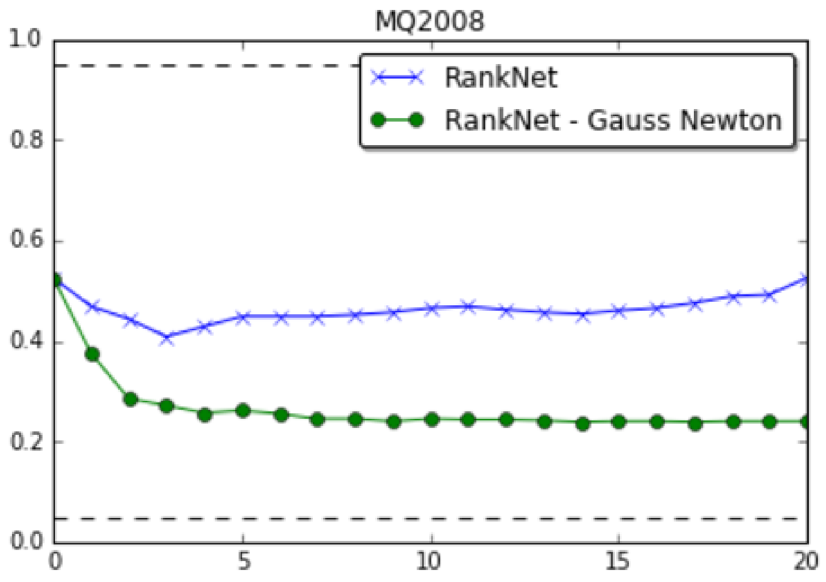


FIGURE 4.9: Percentage of Misordered Pairs for MQ2008 Data Set

Data set: MQ2007

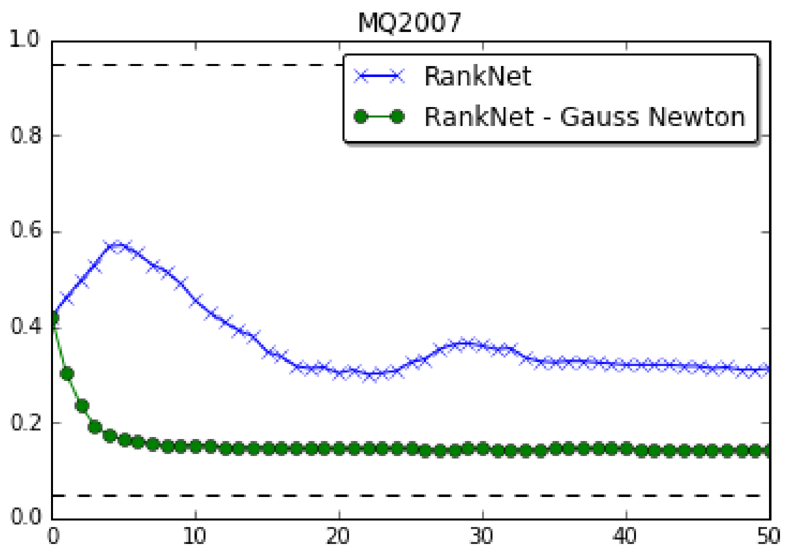


FIGURE 4.10: Percentage of Misordered Pairs for MQ2007 Data Set

Data set: Sylva

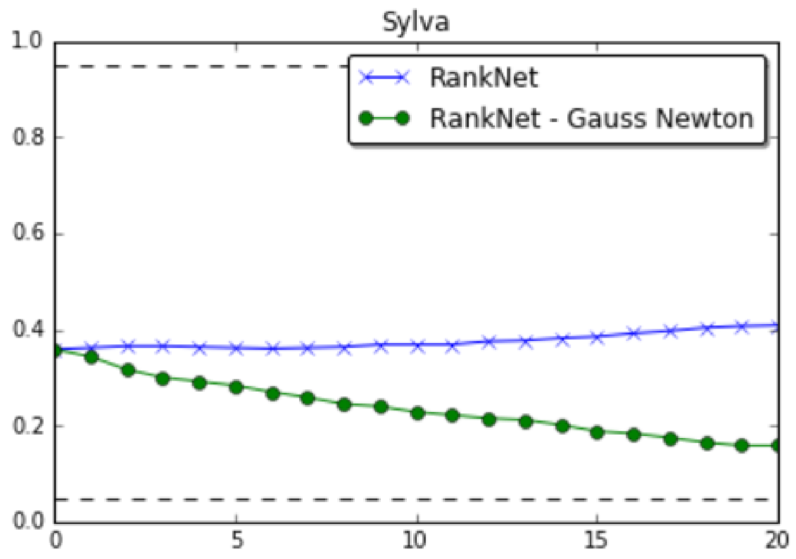


FIGURE 4.11: Percentage of Misordered Pairs for Sylva Data Set

Data set: HIVA

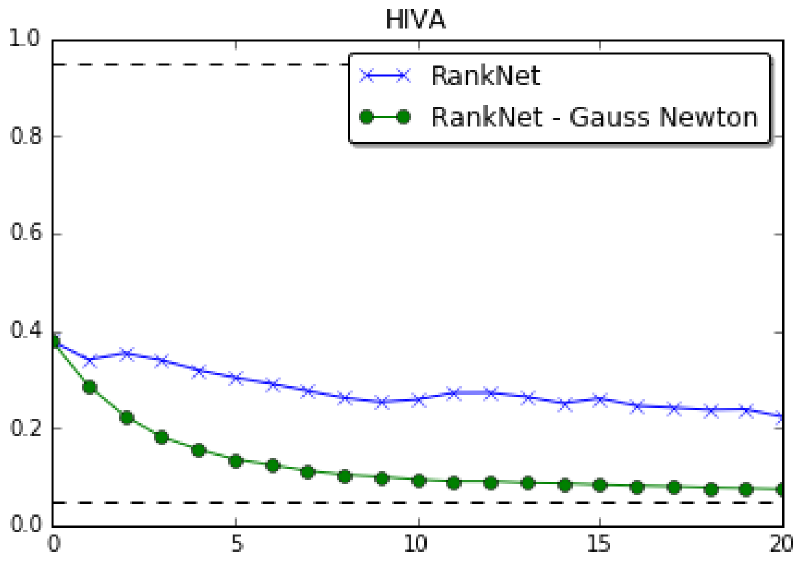


FIGURE 4.12: Percentage of Misordered Pairs for HIVA Data Set

Data set: NOVA

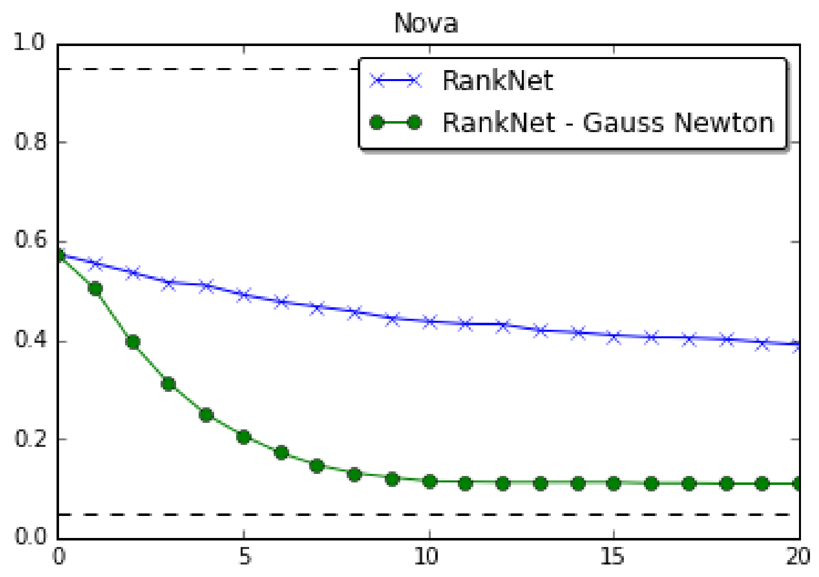


FIGURE 4.13: Percentage of Misordered Pairs for NOVA Data Set

The following table illustrates the accuracy on the training sets after training with RankNet and Gauss-Newton approximation with RankNet algorithm.

Percentage of Misordered Pairs		
Data set	RankNet	RankNet with Gauss Newton
MQ2008	52%	24%
MQ2007	31%	14%
SYLVA	40%	15%
HIVA	22%	7%
NOVA	39%	10%

4.3 Training with Constrained Optimization

All the previous results were obtained by applying first and second-order optimization methods to minimize the cost function. However, no constraints were imposed in the problem. The output of the artificial neural network will hardly change for large parameters values since the output of the sigmoid function for each neuron will fall in the saturation region this phenomenon is called network paralysis [18]. To show that analytically note that the derivative of the sigmoid function goes to zero for large input values as follows:

$$\begin{aligned} a &= \sigma(z) \\ a' &= \sigma'(z) \\ &= \sigma(z)(1-\sigma(z)) \\ &= -\frac{\sigma(z)^2}{\exp(z)} \longrightarrow 0 \text{ as } z \longrightarrow 0 \end{aligned}$$

The following figure illustrates how the derivative of the sigmoid function vanishes as the region expand to infinity:

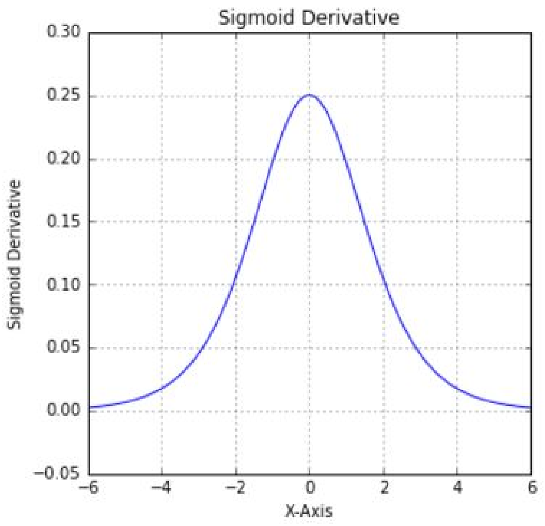


FIGURE 4.14: Sigmoid Function Derivative

Algorithms 2 and 3 assumes no constraints on the parameters space. This might lead to the network paralysis phenomena. To avoid the network paralysis, we impose constraints on the parameters space to stay in the linear region of the sigmoid function output. The linear region of the sigmoid function provides the highest change of the ANN output for any small perturbation of the parameters.
We will choose the bounds to be $[-0.7, 0.7]$ this selection can be justified be visualizing the output of the sigmoid function with respect to its input and the approximated linear region of the sigmoid function output.
Limiting the values fed to the sigmoid functions to stay in the linear region impose

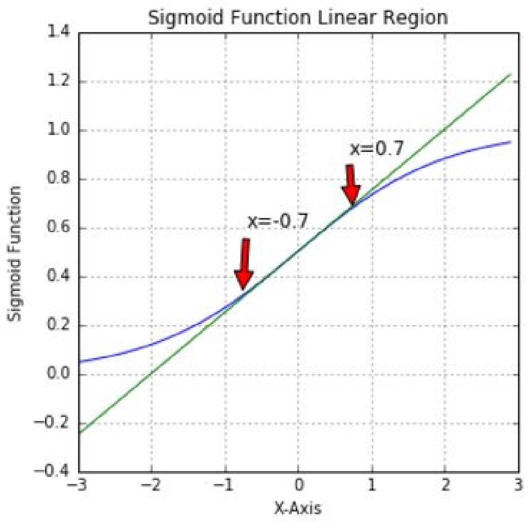


FIGURE 4.15: Sigmoid Function Linear Region

inequality constraints to the optimization problem as follows:

$$\begin{aligned} \min_{\theta} \quad & C(\theta) \\ \text{subject to} \quad & -0.7 \leq z^l \leq 0.7 \end{aligned}$$

The constraints of the above problem can be rewritten in the following matrix form:

$$\begin{aligned} \min_{\theta} \quad & C(\theta) \\ \text{subject to} \quad & \Lambda\theta - B + s = 0 \\ & -s \leq 0 \end{aligned} \tag{4.8}$$

where s is a slack variable and Λ, B are the parameters matrices for the case of a single hidden layer:

$$\Lambda = \begin{bmatrix} a^{0^T} & 1 & 0 & 0 \\ -a^{0^T} & -1 & 0 & 0 \\ 0 & 0 & a^{1^T} & 1 \\ 0 & 0 & -a^{1^T} & -1 \end{bmatrix} \theta = \begin{bmatrix} \theta^1 \\ \theta_b^1 \\ \theta^2 \\ \theta_b^2 \end{bmatrix} B = \begin{bmatrix} 0.7 \\ 0.7 \\ 0.7 \\ 0.7 \end{bmatrix}$$

Then the original problem can be approximated by the following optimization problem:

$$\begin{aligned} \min_{\theta} \quad & C(\theta) + \nabla C(\theta)^T \delta\theta + \frac{1}{2}\delta\theta^T \hat{H(\theta)}\delta\theta \\ \text{subject to} \quad & \Lambda\theta - B + s = 0 \\ & -s \leq 0 \end{aligned} \tag{4.9}$$

In an unconstrained optimization Newton method computed $\triangle\theta$ for the problem (4.1) as $\triangle\theta = -H(\theta)^{-1}\nabla C(\theta)$ or using the Gauss-Newton matrix approximation. $\triangle\theta = -\hat{H(\theta)}^{-1}\nabla C(\theta)$. However, this approach will not guarantee the optimality of the solution to the problem (4.9) due to the constraints imposed.
The optimal θ that minimizes problem (4.9) is computed using the theory of quadratic programming [9]. The solution is feasible only if the Hessian matrix H is a positive semi-definite matrix and hence invertible which is not the case in ANN. The theory of Gauss-Newton approximation guarantees the positive semi-definiteness of the approximation $\hat{H(\theta)}$ hence we can use the theory of quadratic programming with inequality constraints to solve the problem (4.9) if we substitute the Hessian matrix by the Gauss-Newton matrix.

Chapter 5

Conclusion

ANN is a powerful model that can be used in many applications. The complexity of the ANN model makes it hard to estimate the optimal parameters. To overcome this obstacle, we used the Gauss-Newton matrix that approximates the Hessian matrix without the need for direct computation of the second order derivatives. The training algorithm proposed in the previous chapter shows that second order convex optimization techniques, like Newton's method, are superior to the traditional back propagation algorithm for ANN training. The results indicate that using Gauss-Newton matrix increase the accuracy and the speed of training. Such results are major for applications that usually suffer from long training times such as ranking.

Also, a new parameters estimation technique can use the Gauss-Newton approximation in the initial iterations of the training followed by the traditional back propagation algorithms for the rest of iterations. The benefit of this technique is it combines the speed of the Gauss-Newton approximation at initial iterations where their parameters are far from their optimal values, and the back propagation algorithm for fine tuning the parameters.

Training ANN with Gauss-Newton method updates the parameters with a Newton's step which moves the parameters faster to the region near the minimum value. In recent research [11], it is shown that in high-dimensional problems, such as DNN, the training near saddle points slows down the learning dramatically. A proposed second order optimization algorithm is used to overcome the saddle points. Therefore, in the future, we believe that our proposed algorithm can be generalized to overcome the saddle points issue for DNN.

Appendix A: Hessian Matrix Derivation

Mixed derivatives for θ^2 will compute the first column of the Hessian matrix H^1

$$\begin{aligned}\frac{\partial^2 C}{\partial\theta^2\partial\theta^2} &= \frac{\partial\delta^2 a^1}{\partial\theta^2} = \delta^2\frac{\partial a^1}{\partial\theta^2} + a^1\frac{\partial\delta^2}{\partial\theta^2}\\ &= a^1\frac{\partial(\frac{\partial C}{\partial\hat{Y}}\sigma'(z^2))}{\partial\theta^2} = a^1(\frac{\partial C}{\partial\hat{Y}}\frac{\partial\sigma'(z^2)}{\partial\theta^2} + \sigma'(z^2)\frac{\partial\frac{\partial C}{\partial\hat{Y}}}{\partial\theta^2})\\ &= a^1(\frac{\partial C}{\partial\hat{Y}}\sigma''(z^2){a^1}^T + \sigma'(z^2)\frac{\partial\frac{\partial C}{\partial\hat{Y}}}{\partial\theta^2})\end{aligned}$$

$$\begin{aligned}\frac{\partial^2 C}{\partial\theta_b^2\partial\theta^2} &= \frac{\partial\delta^2 a^1}{\partial\theta_b^2} = \delta^2\frac{\partial a^1}{\partial\theta_b^2} + a^1\frac{\partial\delta^2}{\partial\theta_b^2}\\ &= a^1\frac{\partial(\frac{\partial C}{\partial\hat{Y}}\sigma'(z^2))}{\partial\theta_b^2} = a^1(\frac{\partial C}{\partial\hat{Y}}\frac{\partial\sigma'(z^2)}{\partial\theta_b^2} + \sigma'(z^2)\frac{\partial\frac{\partial C}{\partial\hat{Y}}}{\partial\theta_b^2})\\ &= a^1(\frac{\partial C}{\partial\hat{Y}}\sigma''(z^2) + \sigma'(z^2)\frac{\partial\frac{\partial C}{\partial\hat{Y}}}{\partial\theta_b^2})\end{aligned}$$

$$\begin{aligned}\frac{\partial^2 C}{\partial\theta^1\partial\theta^2} &= \frac{\partial\delta^2 a^1}{\partial\theta^1} = \delta^2\frac{\partial a^1}{\partial\theta^1} + a^1\frac{\partial\delta^2}{\partial\theta^1}\\ &= \frac{\partial C}{\partial\hat{Y}}\sigma'(z^2)\sigma'(z^1){a^0}^T + a^1\frac{\partial(\frac{\partial C}{\partial\hat{Y}}\sigma'(z^2))}{\partial\theta^1}\\ &= \frac{\partial C}{\partial\hat{Y}}\sigma'(z^2)\sigma'(z^1){a^0}^T + a^1(\frac{\partial C}{\partial\hat{Y}}\sigma''(z^2){a^0}^T\theta^2 + \sigma'(z^2)\frac{\partial\frac{\partial C}{\partial\hat{Y}}}{\partial\theta^1})\end{aligned}$$

$$\begin{aligned}\frac{\partial^2 C}{\partial\theta_b^1\partial\theta^2} &= \frac{\partial\delta^2 a^1}{\partial\theta_b^1} = \delta^2\frac{\partial a^1}{\partial\theta_b^1} + a^1\frac{\partial\delta^2}{\partial\theta_b^1}\\ &= \frac{\partial C}{\partial\hat{Y}}\sigma'(z^2)\sigma'(z^1) + a^1\frac{\partial(\frac{\partial C}{\partial\hat{Y}}\sigma'(z^2))}{\partial\theta_b^1}\\ &= \frac{\partial C}{\partial\hat{Y}}\sigma'(z^2)\sigma'(z^1) + a^1(\frac{\partial C}{\partial\hat{Y}}\sigma''(z^2)\theta^2 + \sigma'(z^2)\frac{\partial\frac{\partial C}{\partial\hat{Y}}}{\partial\theta_b^1})\end{aligned}$$

Mixed derivatives for θ_b^2 will compute the second column of the Hessian matrix H^2.

$$\begin{aligned}\frac{\partial^2 C}{\partial\theta^2\partial\theta_b^2} &= \frac{\partial\delta^2}{\partial\theta^2} = \frac{\partial C}{\partial\hat{Y}}\sigma''(z^2){a^1}^T + \sigma'(z^2)\frac{\partial\frac{\partial C}{\partial\hat{Y}}}{\partial\theta^2}\\ \frac{\partial^2 C}{\partial\theta_b^2\partial\theta_b^2} &= \frac{\partial\delta^2}{\partial\theta_b^2} = \frac{\partial C}{\partial\hat{Y}}\sigma''(z^2) + \sigma'(z^2)\frac{\partial\frac{\partial C}{\partial\hat{Y}}}{\partial\theta_b^2}\\ \frac{\partial^2 C}{\partial\theta^1\partial\theta_b^2} &= \frac{\partial\delta^2}{\partial\theta^1} = \frac{\partial C}{\partial\hat{Y}}\sigma''(z^2){a^0}^T\theta^2 + \sigma'(z^2)\frac{\partial\frac{\partial C}{\partial\hat{Y}}}{\partial\theta^1}\\ \frac{\partial^2 C}{\partial\theta_b^1\partial\theta_b^2} &= \frac{\partial\delta^2}{\partial\theta_b^1} = \frac{\partial C}{\partial\hat{Y}}\sigma''(z^2)\theta^2 + \sigma'(z^2)\frac{\partial\frac{\partial C}{\partial\hat{Y}}}{\partial\theta_b^1}\end{aligned}$$

Mixed derivatives for θ^1 will compute the third column of the Hessian matrix H^3.

$$\begin{aligned}
\frac{\partial^2 C}{\partial\theta^2\partial\theta^1} &= \frac{\partial\delta^1 a^0}{\partial\theta^2} = \sigma'(z^1)a^0\\
\frac{\partial^2 C}{\partial\theta_b^2\partial\theta^1} &= \frac{\partial\delta^1 a^0}{\partial\theta_b^2} = 0\\
\frac{\partial^2 C}{\partial\theta^1\partial\theta^1} &= \frac{\partial\delta^1 a^0}{\partial\theta^1} = \theta^2\sigma''(z^1){a^0}^T a^0\\
\frac{\partial^2 C}{\partial\theta_b^1\partial\theta^1} &= \frac{\partial\delta^1 a^0}{\partial\theta_b^1} = \theta^2\sigma''(z^1)a^0
\end{aligned}$$

Mixed derivatives for θ_b^1 will compute the third column of the Hessian matrix H^4.

$$\begin{aligned}
\frac{\partial^2 C}{\partial\theta^2\partial\theta_b^1} &= \frac{\partial\delta^1}{\partial\theta^2} = \sigma'(z^1)\\
\frac{\partial^2 C}{\partial\theta_b^2\partial\theta_b^1} &= \frac{\partial\delta^1}{\partial\theta_b^2} = 0\\
\frac{\partial^2 C}{\partial\theta^1\partial\theta_b^1} &= \frac{\partial\delta^1}{\partial\theta^1} = \theta^2\sigma''(z^1){a^0}^T\\
\frac{\partial^2 C}{\partial\theta_b^1\partial\theta_b^1} &= \frac{\partial\delta^1}{\partial\theta_b^1} = \theta^2\sigma''(z^1)
\end{aligned}$$

Appendix B: RankNet Algorithm

To apply the back propagation algorithm we need to find the partial derivatives of the cost function with respect to the parameters θ^l and θ_b^l for all the layers l. Recalling our discussion in Chapter 2 for deriving the gradient of the cost function C with respect to all the parameters of the neural network.

$$\frac{\partial C}{\partial \theta^2} = \delta^2 a^1$$
$$\frac{\partial C}{\partial \theta_b^2} = \delta^2$$
$$\frac{\partial C}{\partial \theta^1} = \delta^1 a^0$$
$$\frac{\partial C}{\partial \theta_b^1} = \delta^1$$

RankNet modifies the back propagation algorithm to generalize the learning into pair of samples $\hat{\mathrm{r}}_i$ and $\hat{\mathrm{r}}_j$. Let r_{ij} be the relative rank between the two samples and let $\hat{\mathrm{r}}_{ij}$ be the relative rank between $\hat{\mathrm{r}}_i, \hat{\mathrm{r}}_j$ that can be modeled as

$$\hat{\mathrm{r}}_{ij} = \gamma(\hat{\mathrm{r}}_i - \hat{\mathrm{r}}_j)$$

Let the cost function $C(\hat{\mathrm{r}}_{ij}, r_{ij})$ be the Cross-Entropy loss between the relative rank target probability $\overline{p}$ and the relative rank posterior probability p_{ij}.

$$C \equiv -\overline{p}_{ij} \log p_{ij} - (1 - \overline{p}_{ij}) \log(1 - p_{ij})$$

By substituting the p_{ij} term from equation (2.1) in equation (4.9).

$$C = -\overline{p}_{ij}\hat{\mathrm{r}}_{ij} + \log(1 + \exp(\hat{\mathrm{r}}_{ij}))$$

Note that the Cross-Entropy cost function used in the RankNet algorithm is convex. However it may not have a zero cost minimum and it is unbounded. In order to derive a learning rule we will compute the derivative of the cost function C with respect to the model parameters θ using chain rule.

$$\frac{\partial C}{\partial \theta} = \frac{\partial C}{\partial \hat{\mathrm{r}}_i}\frac{\partial \hat{\mathrm{r}}_i}{\partial \theta} - \frac{\partial C}{\partial \hat{\mathrm{r}}_j}\frac{\partial \hat{\mathrm{r}}_j}{\partial \theta}$$

Note that the partial derivative of the cost function C with respect to $\hat{\mathrm{r}}_i, \hat{\mathrm{r}}_j$ can be computed directly by differentiating equation (4.10). An interesting result is that both derivatives are equal except a negative sign.

$$\begin{aligned}\frac{\partial C}{\partial \hat{\mathrm{r}}_i} &= -\overline{p}_{ij}\frac{\partial \hat{\mathrm{r}}_{ij}}{\partial \hat{\mathrm{r}}_i} + \frac{\partial}{\partial \hat{\mathrm{r}}_i} log(1+\exp(\hat{\mathrm{r}}_{ij})) = -\overline{p}_{ij}\frac{\partial \hat{\mathrm{r}}_{ij}}{\partial \hat{\mathrm{r}}_i} + \frac{\exp(\hat{\mathrm{r}}_{ij})}{1+\exp(\hat{\mathrm{r}}_{ij})}\frac{\partial \hat{\mathrm{r}}_{ij}}{\partial \hat{\mathrm{r}}_i} \\ &= \frac{\partial \hat{\mathrm{r}}_{ij}}{\partial \hat{\mathrm{r}}_i}\left(\frac{\exp(\hat{\mathrm{r}}_{ij})}{1+\exp(\hat{\mathrm{r}}_{ij})} - \overline{p}_{ij}\right) = \gamma\left(p_{ij} - \overline{p}_{ij}\right) \\ &= -\frac{\partial C}{\partial \hat{\mathrm{r}}_j} = \pi_{ij}\end{aligned}$$

This result simplifies equation (4.11) to

$$\frac{\partial C}{\partial \theta} = \pi_{ij}\left(\frac{\partial \hat{\mathrm{r}}_i}{\partial \theta} - \frac{\partial \hat{\mathrm{r}}_j}{\partial \theta}\right)$$

This learning rule defined by the RankNet algorithm modifies the back propagation algorithm to derive the gradient of the cost function C which is defined as the Cross-Entropy loss for the relative rank estimates between any two samples with respect to the model parameters θ.

Appendix C: Implementation

```python
    def backward(self, a1, a2):
        r12 = self.gamma * (a1[self.L] - a2[self.L])
        p12 = self.sigmoid(r12); p12_par = self.sigmoid(self.r12_par)
        s12 = 2 * p12_par -1
        pi = - self.gamma * self.sigmoid(-r12)
        deltas_1 = pi; deltas_2 = -pi
        for i in range(self.L - 1, -1, -1):
            deltas_prev_1 = np.multiply(np.multiply(
            np.dot(self.thetas[i].T, deltas_1) , a1[i]), 1 - a1[i])
            deltas_prev_2 = np.multiply(np.multiply(
            np.dot(self.thetas[i].T, deltas_2) , a2[i]), 1 - a2[i])
            gradient, S = self.compute_step(i, a1, a2,
            deltas_1, deltas_2, pi, p12)
            self.weight_update(i, gradient, S)
            deltas_1 = deltas_prev_1[1:, :]
            deltas_2 = deltas_prev_2[1:, :]
        return None

    def compute_step(self, i, a1, a2, deltas_1, deltas_2, pi, p12):
        gradient_1 = np.sum(np.array([np.outer(deltas_1[:, k],
        a1[i][:, k]) for k in range(deltas_1.shape[1])]), axis=0)
        gradient_2 = np.sum(np.array([np.outer(deltas_2[:, k],
        a2[i][:, k]) for k in range(deltas_2.shape[1])]), axis=0)
        gradient = float(pi)* (gradient_1 - gradient_2)
        H_1 = np.dot(np.transpose(gradient_1), gradient_1)
        H_2 = np.dot(np.transpose(gradient_2), gradient_2)
        H = self.gamma * float(pi) * float(p12) *
        np.dot(np.transpose(gradient), gradient) + float(pi)*(H_2 - H_1)
        S = np.transpose(np.dot(pinv(np.matrix(H)),
        np.transpose(gradient)))
        return gradient, S

    def weight_update(self, i, gradient, S):
        self.momentum[i] = self.mu*self.momentum[i] + S
        self.thetas[i] = self.thetas[i] - self.alpha *
        self.momentum[i] - self.beta * self.thetas[i]
        return None
```

Listing 1: RankNet Back Propagation

```python
    def train_neural_network(self, method, niter, mu, alpha, beta):
        self.mu = mu
        self.alpha = alpha
        self.beta = beta
        self.method = method

        cost = [self.countMisorderedPairs(self.X, self.pairs)]

        start = time.time()
        i = 0; Stopping_Criteria = 10
        while(np.abs(Stopping_Criteria) >= self.tol):
            for pair in self.pairs:
                s1 = np.array(self.X.iloc[pair[0]]).reshape((1,-1))
                s2 = np.array(self.X.iloc[pair[1]]).reshape((1,-1))
                a1 = self.forward(s1)
                a2 = self.forward(s2)
                self.backward(a1, a2)
            Stopping_Criteria = self.countMisorderedPairs(self.X, self.pairs)
            cost.append(Stopping_Criteria)
            i = i + 1
            if(i >= niter):
                break;

        m, s = divmod(time.time()-start, 60)
        print('Training took: ', m, 'm', np.round(s,2), 's')

        return cost, i

    def predict_neural_network(self, X, thetas):
        self.thetas = thetas
        return self.forward(X)[self.L][0].T
```

Listing 2: Train and Predict

References

[1] Abu-Mostafa, Y. S., Magdon-Ismail, M., and Lin, H. (2012). *Learning from data: A short course.* United States: AMLBook.com.

[2] Assaad, M., Boné, R., and Cardot, H. (2008). A new boosting algorithm for improved time-series forecasting with recurrent neural networks. *Information Fusion, 9* (1), 41-55. doi:10.1016/j.inffus.2006.10.009

[3] Boyd, S. P., and Vandenberghe, L. (2004). *Convex optimization.* Cambridge, UK: Cambridge University Press.

[4] Breiman, Leo. "Bagging Predictors". *Mach Learn 24.2* (1996): 123-140. Web.

[5] Burges, C. J. (2010). *From ranknet to lambdarank to lambdamart: An overview* (Tech. No. MSR-TR-2010-82).

[6] Burges, C., Shaked, T., Renshaw, E., Lazier, A., Deeds, M., Hamilton, N., and Hullender, G. (2005). Learning to rank using gradient descent. *Proceedings of the 22nd International Conference on Machine Learning - ICML '05.* doi:10.1145/1102351.1102363

[7] Chen, P. (2011). Hessian Matrix vs. Gauss–Newton Hessian Matrix. *SIAM Journal on Numerical Analysis SIAM J. Numer. Anal., .* 49(4), 1417-1435. doi:10.1137/100799988

[8] Cohen, S. (2016). *Bayesian Analysis in Natural Language Processing.* Morgan and Claypool.

[9] Crammer, K., and Singer, Y. (2001). Pranking with ranking. *Advances in Neural Information Processing Systems.*

[10] Cybenko, G. (1989, December). Approximation by superpositions of a sigmoidal function. *Mathematics of Control, Signals and Systems, 2(4),* 303-314. doi:10.1007/BF02551274

[11] Dauphin, Y. N., Pascanu, R., Gulcehre, C., Cho, K., Ganguli, S., and Bengio, Y. (2014). Identifying and attacking the saddle point problem in high-dimensional non-convex optimization. *Advances in Neural Information Processing Systems,* 2933-2941.

[12] Dshalalow, J. H. (2001). *Real analysis: An introduction to the theory of real functions and integration.* Boca Raton: Chapman and Hall/CRC.

[13] Duch, W., and Jankowski, N. (2000). Taxonomy of neural transfer functions. Proceedings of the IEEE-INNS-ENNS *International Joint Conference on Neural Networks. IJCNN 2000. Neural Computing: New Challenges and Perspectives for the New Millennium.*

[14] Gentle, J. E., Hardle, W., and Mori, Y. (2012). *Handbook of computational statistics: Concepts and methods.* Heidelberg: Springer

[15] Gupta, M. M., Jin, L., and Homma, N. (2003). *Static and dynamic neural networks: From fundamentals to advanced theory.* New York: Wiley.

[16] Jordan, M. I. (1995). *Why the logistic function? A tutorial discussion on probabilities and neural networks* (Tech. No. 9503).

[17] Polyak, B. (1964). Some methods of speeding up the convergence of iteration methods. *USSR Computational Mathematics and Mathematical Physics, 4* (5), 1-17. doi:10.1016/0041-5553(64)90137-5

[18] Wasserman, P. D. (1989). *Neural computing: Theory and practice.* New York: Van Nostrand Reinhold.

Vita

Baha Khalil was born on July 5, 1988, in Sharjah, in the United Arab Emirates. He graduated from the American University of Sharjah, in 2011. His degree was a Bachelor of Science in Electrical Engineering.
Baha Khalil worked as a technical support engineer and as a business intelligence specialist for four years at Schneider Electric. Later joined Thomson Reuters to work as a data scientist. In 2012, Baha Khalil began a Master's program in Mathematics at the American University of Sharjah.